XENIX® System V

Development System

Programmer's Guide

This document was typeset with an IMAGEN® 8/300 Laser Printer.

SCO Document Number: XG-5-1-86-3.0

Contents

1 **Introduction**

1.1 Overview 1-1
1.2 Creating Programs 1-1
1.3 Creating and Maintaining Libraries 1-2
1.4 Maintaining Program Source Files 1-2
1.5 Creating Programs With Shell Commands 1-3
1.6 About This Guide 1-3
1.7 Notational Conventions 1-4

2 **make: A Program Maintainer**

2.1 Introduction 2-1
2.2 Creating a Makefile 2-1
2.3 Invoking make 2-3
2.4 Using Pseudo-Target Names 2-5
2.5 Using Macros 2-5
2.6 Using Shell Environment Variables 2-8
2.7 Using the Built-In Rules 2-9
2.8 Changing the Built-in Rules 2-10
2.9 Using Libraries 2-12
2.10 Troubleshooting 2-13
2.11 Using make: An Example 2-14

3 **SCCS: A Source Code Control System**

3.1 Introduction 3-1
3.2 Basic Information 3-1
3.3 Creating and Using S-files 3-5
3.4 Using Identification Keywords 3-14
3.5 Using S-file Flags 3-16
3.6 Modifying S-file Information 3-18
3.7 Printing from an S-file 3-21
3.8 Editing by Several Users 3-22
3.9 Protecting S-files 3-24
3.10 Repairing SCCS Files 3-26
3.11 Using Other Command Options 3-28

4 **lint: A C Program Checker**

4.1 Introduction 4-1
4.2 Invoking lint 4-1
4.3 Options 4-2

4.4 Checking for Unused Variables and Functions 4-3
4.5 Checking Local Variables 4-4
4.6 Checking for Unreachable Statements 4-5
4.7 Checking for Infinite Loops 4-6
4.8 Checking Function Return Values 4-6
4.9 Checking for Unused Return Values 4-7
4.10 Checking Types 4-7
4.11 Checking Type Casts 4-8
4.12 Checking for Nonportable Character Use 4-9
4.13 Checking for Assignment of longs to ints 4-9
4.14 Checking for Strange Constructions 4-9
4.15 Checking for Use of Older C Syntax 4-11
4.16 Checking Pointer Alignment 4-12
4.17 Checking Expression Evaluation Order 4-12
4.18 Embedding Directives 4-13
4.19 Checking For Library Compatibility 4-14

5 lex: A Lexical Analyzer

5.1 Introduction 5-1
5.2 lex Regular Expressions 5-3
5.3 Invoking **lex** 5-4
5.4 Specifying Character Classes 5-4
5.5 Specifying an Arbitrary Character 5-5
5.6 Specifying Optional Expressions 5-5
5.7 Specifying Repeated Expressions 5-6
5.8 Specifying Alternation and Grouping 5-6
5.9 Specifying Context Sensitivity 5-6
5.10 Specifying Expression Repetition 5-7
5.11 Specifying Definitions 5-7
5.12 Specifying Actions 5-8
5.13 Handling Ambiguous Source Rules 5-12
5.14 Specifying Left Context Sensitivity 5-15
5.15 Specifying Source Definitions 5-17
5.16 lex and yacc 5-19
5.17 Specifying Character Sets 5-23
5.18 Source Format 5-24

6 yacc: A Compiler-Compiler

6.1 Introduction 6-1
6.2 Specifications 6-4
6.3 Actions 6-6
6.4 Lexical Analysis 6-9
6.5 How the Parser Works 6-11
6.6 Ambiguity and Conflicts 6-16
6.7 Precedence 6-21
6.8 Error Handling 6-23

6.9 The yacc Environment 6-26
6.10 Preparing Specifications 6-27
6.11 Input Style 6-27
6.12 Left Recursion 6-28
6.13 Lexical Tie-ins 6-29
6.14 Handling Reserved Words 6-30
6.15 Simulating Error and Accept in Actions 6-30
6.16 Accessing Values in Enclosing Rules 6-30
6.17 Supporting Arbitrary Value Types 6-31
6.18 A Small Desk Calculator 6-32
6.19 yacc Input Syntax 6-35
6.20 An Advanced Example 6-38
6.21 Old Features 6-44

7 Using Signals

7.1 Introduction 7-1
7.2 Using the signal Function 7-1
7.3 Catching Several Signals 7-7
7.4 Controlling Execution With Signals 7-8
7.5 Using Signals in Multiple Processes 7-12

A M4: A Macro Processor

A.1 Introduction A-1
A.2 Invoking m4 A-1
A.3 Defining Macros A-2
A.4 Quoting A-3
A.5 Using Arguments A-5
A.6 Using Arithmetic Built-ins A-6
A.7 Manipulating Files A-7
A.8 Using System Commands A-8
A.9 Using Conditionals A-8
A.10 Manipulating Strings A-9
A.11 Printing A-11

B XENIX System Calls

B.1 Introduction B-1
B.2 Executable File Format B-1
B.3 Revised System Calls B-1
B.4 Version 7 Additions B-3
B.5 Changes to the ioctl Function B-3
B.6 Pathname Resolution B-3
B.7 Using the mount () and chown () Functions B-4
B.8 Super-Block Format B-4
B.9 Separate Version Libraries B-4

Chapter 1
Introduction

1.1 Overview 1-1

1.2 Creating Programs 1-1

1.3 Creating and Maintaining Libraries 1-2

1.4 Maintaining Program Source Files 1-2

1.5 Creating Programs With Shell Commands 1-3

1.6 About This Guide 1-3

1.7 Notational Conventions 1-4

1.1 Overview

This guide explains how to use the XENIX Development system to create and maintain C language and assembly language programs. The system provides a broad spectrum of programs and commands to help you design and develop applications and system software. These programs and commands enable you to create C and assembly language programs for execution on the XENIX system. They also let you debug these programs, automate their creation, and maintain different versions of the programs you develop.

The following sections introduce the programs and commands of the XENIX Development System, and explain the steps you can take to develop programs for the XENIX system. Most of the programs and commands in these introductory sections are fully explained later in this guide. Some commands mentioned here are part of the XENIX Operating System. These are explained in the XENIX *User's Guide* and XENIX *Operations Guide.*

1.2 Creating Programs

The C programming language can meet the needs of most programming projects. A complete description of how to write, compile, link, and run C programs under the XENIX operating system is provided in three documents: the *C User's Guide*, the *C Language Reference*, and the *C Library Reference*.

You may also create assembly language programs using **masm**(CP), the XENIX assembler. **masm** assembles source files and produces relocatable object files that can be linked to your C language programs with **ld**(CP). The **ld** program is the XENIX linker. It links relocatable object files created by the C compiler or assembler to produce executable programs. Note that the **cc**(CP) command automatically invokes the linker and the assembler, so use of either **masm** or **ld** is optional.

You can create source files for lexical analyzers and parsers using the program generators **lex**(CP) and **yacc**(CP). Lexical analyzers are used in programs to pick patterns out of complex input and convert these patterns into meaningful values or tokens. Parsers are used in programs to convert meaningful sequences of tokens and values into actions. The **lex** program is the XENIX lexical analyzer generator. It generates lexical analyzers, written in C program statements, from given specification files. The **yacc** program is the XENIX parser generator. It generates parsers, written in C program statements, from given specification files. **lex** and **yacc** are often used together to make complete programs.

You can preprocess C and assembly language source files, or even **lex** and **yacc** source files using the **m4**(CP) macro processor. The **m4** program performs several preprocessing functions, such as converting macros to their defined values and including the contents of files into a source file.

1.3 Creating and Maintaining Libraries

You can create libraries of useful C and assembly language functions and programs using the **ar** and **ranlib**(CP) programs. **ar**(CP), the XENIX archiver, can be used to create libraries of relocatable object files. **ranlib**, the XENIX random library generator, converts archive libraries to random libraries and places a table of contents at the front of each library.

The **lorder**(CP) command finds the ordering relation in an object library. The **tsort**(CP) command topologically sorts object libraries so that dependencies are apparent.

1.4 Maintaining Program Source Files

You can automate the creation of executable programs from C and assembly language source files and maintain your source files using the **make** program and the SCCS commands.

The **make** program is the XENIX program maintainer. It automates the steps required to create executable programs, and provides a mechanism for ensuring up-to-date programs. It is used with medium-scale programming projects.

The Source Code Control (SCCS) commands let you maintain different versions of a single program. The commands compress all versions of a source file into a single file containing a list of differences. These commands also restore compressed files to their original size and content.

Many XENIX commands let you carefully examine a program's source files. The **ctags**(CP) command creates a tags file so that C functions can be quickly found in a set of related C source files. The **mkstr**(CP) command creates an error message file by examining a C source file.

Other commands let you examine object and executable binary files. The **nm**(CP) command prints the list of symbol names in a program. The **hd**(C) command performs a hexadecimal dump of given files, printing files in a variety of formats, one of which is hexadecimal. The **size**(CP) command reports the size of an object file. The **strings**(CP) command finds and prints readable text (strings) in an object or other binary file. The **strip**(CP) command removes symbols and relocation bits from executable files. The **sum**(C) command computes a checksum value for a file and a count of its blocks. It is used in looking for bad spots in a file and for verifying transmission of data between systems. The **xstr** command extracts strings from C programs to implement shared strings.

1.5 Creating Programs With Shell Commands

In some cases, it is easier to write a program as a series of XENIX shell commands than it is to create a C language program. Shell commands provide much of the same control capability as the C language, and give direct access to all the commands and programs normally available to the XENIX user.

The **csh**(C) command invokes the C-shell, a XENIX command interpreter. The C-shell interprets and executes commands taken from the keyboard or from a command file. It has a C-like syntax which makes programming in this command language easy. It also has an aliasing facility, and a command history mechanism.

1.6 About This Guide

This guide is intended for programmers who are familiar with the C programming language and with the XENIX system. It is organized as follows:

Chapter 1, "Introduction," introduces the XENIX software development programs provided with this package and summarizes the organization of this guide and the conventions used.

Chapter 2, "make: A Program Maintainer," explains how to automate the development of a program or other project using the **make** program.

Chapter 3, "SCCS: A Source Code Control System," explains how to control and maintain all versions of a project's source files using the SCCS commands.

Chapter 4, "lint: A C Program Checker," describes the XENIX program checker, **lint**, and describes the available options.

Chapter 5, "lex: A Lexical Analyzer," explains how to create lexical analyzers using the program generator **lex**.

Chapter 6, "yacc: A Compiler-Compiler," explains how to create parsers using the program generator **yacc**.

Chapter 7, "Using Signals," describes the signal functions. These functions let a program process signals that are normally processed by the system.

Appendix A, "m4: A Macro Processor," explains how to use, create and process macros using the **m4** C functions.

Appendix B, "XENIX System Calls," explains how to create and use new XENIX system calls.

C language programmers should read the *C User's Guide* for an explanation of how to compile and debug C language programs.

Assembly language programmers should read the *Macro Assembler User's Guide* for an explanation of the XENIX assembler and Chapter 4, "adb" in the *C User's Guide* for an explanation of how to debug programs.

Programmers who wish to automate the compilation process of their programs should read Chapter 2 for an explanation of the **make** program. Programmers who wish to organize and maintain multiple versions of their programs should read Chapter 3 for an explanation of the Source Code Control System (SCCS) commands.

Special project programmers who need a convenient way to produce lexical analyzers and parsers should read Chapters 6 and 7 for explanations of the **lex** and **yacc** program generators.

1.7 Notational Conventions

This guide uses a number of notational conventions to describe the syntax of XENIX commands:

boldface	Boldface indicates a command, option, flag, or program name to be entered as shown.
	Boldface indicates the name of a library routine, global variable, standard type, constant, keyword, or identifier used by the C library. (To find more information on a given library routine consult the "Alphabetized List" in your XENIX *Reference Manual* for the manual page that describes it.)
italics	Italics indicate a filename. This pertains to library include filenames (i.e. *stdio.h*), as well as, other filenames (i.e. */etc/ttys*).
	Italics indicate a placeholder for a command argument. When entering a command , a placeholder must be replaced with an appropriate filename, number, or option.

	Italics indicate a specific identifier, supplied for variables and functions, when mentioned in text.
	Italics indicate user named routines. (User named routines are followed by open and close parentheses, ().)
	Italics indicate emphasized words or phrases in text.
CAPTIALS	Capitals indicate names of environment variables (i.e. TZ and PATH).
SMALL CAPITALS	Small capitals indicate keys and key sequences (i.e. RETURN).
[ ]	Brackets indicate that the enclosed item is optional. If you do not use the optional item, the program selects a default action to carry out.
...	Ellipses indicate that you can repeat the preceding item any number of times.
.	Vertical ellipses indicate that a portion of a program example is omitted.
" "	Quotation marks indicate the first use of a technical term.
	Quotation marks indicate a reference to a word rather than a command.

Chapter 2
make:
A Program Maintainer

2.1 Introduction 2-1

2.2 Creating a Makefile 2-1

2.3 Invoking make 2-3

2.4 Using Pseudo-Target Names 2-5

2.5 Using Macros 2-5

2.6 Using Shell Environment Variables 2-8

2.7 Using the Built-In Rules 2-9

2.8 Changing the Built-in Rules 2-10

2.9 Using Libraries 2-12

2.10 Troubleshooting 2-13

2.11 Using make: An Example 2-14

2.1 Introduction

The **make**(CP) program provides an easy way to automate the creation of medium to large programs. **make** reads commands from a user-defined "makefile" that lists the files to be created, the commands that create them, and the files from which they are created. When you direct **make** to create a program, it verifies that each file on which the program depends is up to date, then creates the program by executing the given commands. If a file is not up to date, **make** updates it before creating the program. **make** updates a program by executing explicitly given commands, or one of the many built-in commands.

This chapter explains how to use **make** to automate medium-sized programming projects. It explains how to create makefiles for each project, and how to invoke **make** for creating programs and updating files. For more details about the program, see **make**(CP) in the XENIX *Reference Manual.*

2.2 Creating a Makefile

A makefile contains one or more lines of text called dependency lines. A dependency line shows how a given file depends on other files and what commands are required to bring a file up to date. A dependency line has the form:

target ... : [*dependent* ...] [; *command* ...]

where *target* is the filename of the file to be updated, *dependent* is the filename of the file on which the target depends, and *command* is the XENIX command needed to create the target file. Each dependency line must have at least one command associated with it, even if it is only the null command (;).

You may give more than one target filename or dependent filename if desired. Each filename must be separated from the next by at least one space. The target filenames must be separated from the dependent filenames by a colon (:). Filenames must be spelled as defined by the XENIX system. Shell metacharacters, such as star (*) and question mark (?), can also be used.

You may give a sequence of commands on the same line as the target and dependent filenames, if you precede each command with a semicolon (;). You can give additional commands on following lines by beginning each line with a tab character. Commands must be given exactly as they would appear on a shell command line. The at sign (@) may be placed in front of a command to prevent **make** from displaying the command before executing it. Shell commands, such as **cd**(C), must appear on single lines; they must not contain the backslash (\) and newline character combination.

You may add a comment to a makefile by starting the comment with a number sign (#) and ending it with a newline character. All characters after the number sign are ignored. Comments may be place at the end of a dependency line if desired. If a command contains a number sign, it must be enclosed in double quotation marks (").

If a dependency line is too long, you can continue it by entering a backslash (\) and a newline character.

The makefile should be kept in the same directory as the given source files. For convenience, the filenames *makefile*, *Makefile*, *s.makefile*, and *s.Makefile* are provided as default filenames. These names are used by **make** if no explicit name is given at invocation. You may use one of these names for your makefile, or choose one of your own. If the filename begins with the *s.* prefix, **make** assumes that it is an SCCS file and invokes the appropriate SCCS command to retrieve the lastest version of the file.

To illustrate dependency lines, consider the following example. A program named *prog* is made by linking three object files, *x.o*, *y.o*, and *z.o*. These object files are created by compiling the C language source files *x.c*, *y.c*, and *z.c*. Furthermore, the files *x.c* and *y.c* contain the line:

```
#include "defs"
```

This means that *prog* depends on the three object files, the object files depend on the C source files, and two of the source files depend on the include file *defs*. You can represent these relationships in a makefile with the following lines:

```
prog: x.o y.o z.o
        cc x.o y.o z.o -o prog
x.o: x.c defs
        cc -c x.c
y.o: y.c defs
        cc -c y.c
z.o: z.c
        cc -c z.c
```

In the first dependency line, *prog* is the target file and *x.o*, *y.o*, and *z.o* are its dependents. The command sequence:

```
cc x.o y.o z.o -o prog
```

on the next line tells how to create *prog* if it is out of date. The program is out of date if any one of its dependents has been modified since *prog* was last created.

The second, third, and fourth dependency lines have the same form, with the *x.o*, *y.o*, and *z.o* files as targets and *x.c*, *y.c*, *z.c*, and *defs* files as dependents. Each dependency line has one command sequence which defines how to update the given target file.

2.3 Invoking make

Once you have a makefile and wish to update and modify one or more target files in the file, you can invoke **make** by typing its name and optional arguments. The invocation has the form

```
make [option]... [macdef]... [target]...
```

where *option* is a program option used to modify program operation, *macdef* is a macro definition used to give a macro a value or meaning, and *target* is the filename of the file to be updated. It must correspond to one of the target names in the makefile. All arguments are optional. If you give more than one argument, you must separate them with spaces.

You can direct **make** to update the first target file in the makefile by typing just the program name. In this case, **make** searches for the files *makefile*, *Makefile*, *s.makefile*, and *s.Makefile* in the current directory, and uses the first one it finds as the makefile. For example, assume that the current makefile contains the dependency lines given in the last section. Then the command

```
make
```

compares the current date of the *prog* program with the current date each of the object files *x.o*, *y.o*, and *z.o*. It recreates *prog* if any changes have been made to any object file since *prog* was last created. It also compares the current dates of the object files with the dates of the four source files *x.c*, *y.c*, *z.c*, or *defs*, and recreates the object files if the source files have changed. It does this before recreating *prog* so that the recreated object files can be used to recreate *prog*. If none of the source or object files have been altered since the last time *prog* was created, **make** announces this fact and stops. No files are changed.

You can direct **make** to update a given target file by giving the filename of the target. For example,

```
make x.o
```

causes **make** to recompile the *x.o* file, if the *x.c* or *defs* files have changed since the object file was last created. Similarly, the command

```
make x.o z.o
```

causes **make** to recompile *x.o* and *z.o* if the corresponding dependents have been modified. **make** processes target names from the command line in a left to right order.

You can specify the name of the makefile you wish **make** to use by giving the **-f** option in the invocation. The option has the form

-f*filename*

where *filename* is the name of the makefile. You must supply a full pathname if the file is not in the current directory. For example, the command

```
make -f makeprog
```

reads the dependency lines of the makefile named **makeprog** found in the current directory. You can direct **make** to read dependency lines from the standard input by giving "-" as the *filename*. **make** reads the standard input until the end-of-file character is encountered.

You may use the program options to modify the operation of the **make** program. The following list describes some of the options.

-p	Prints the complete set of macro definitions and dependency lines in a makefile.
-i	Ignores errors returned by XENIX commands.
-k	Abandons work on the current entry, but continues on other branches that do not depend on that entry.
-s	Executes commands without displaying them.
-r	Ignores the built-in rules.
-n	Displays commands but does not execute them. **make** even displays lines beginning with the at sign (@).
-e	Ignores any macro definitions that attempt to assign new values to the shell's environment variables.
-t	Changes the modification date of each target file without recreating the files.

Note that **make** executes each command in the makefile by passing it to a separate invocation of a shell. Because of this, care must be taken with certain commands (for example, **cd** and shell control commands) that have meaning only within a single shell process; the results are forgotten before the next line is executed. If an error occurs, **make** normally stops the command.

2.4 Using Pseudo-Target Names

It is often useful to include dependency lines that have pseudo-target names, i.e., names for which no files actually exist or are produced. Pseudo-target names allow **make** to perform tasks not directly connected with the creation of a program, such as deleting old files or printing copies of source files. For example, the following dependency line removes old copies of the given object files when the pseudo-target name "cleanup" is given in the invocation of **make**.

```
cleanup:
        rm x.o y.o z.o
```

Since no file exists for a given pseudo-target name, the target is always assumed to be out of date. Thus, the associated command is always executed.

make also has built-in pseudo-target names that modify its operation. The pseudo-target name ".IGNORE" causes **make** to ignore errors during execution of commands, allowing **make** to continue after an error. This is the same as the **-i** option. (**make** also ignores errors for a given command if the command string begins with a hyphen (-).)

The pseudo-target name ".DEFAULT," defines the commands to be executed either when no built-in rule or a user-defined dependency line exists for the given target. You may give any number of commands with this name. If ".DEFAULT" is not used, and an undefined target is given, **make** prints a message and stops.

The pseudo-target name ".PRECIOUS" prevents dependents of the current target from being deleted when **make** is terminated using the INTERRUPT or QUIT key, and the pseudo-target name ".SILENT" has the same effect as the **-s** option.

2.5 Using Macros

An important feature of a makefile is that it can contain macros. A macro is a short name that represents a filename or command option. The macros can be defined when you invoke **make**, or in the makefile itself.

A macro definition is a line containing a name, an equal sign (=), and a value. The equal sign must not be preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped.) The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly
LIBES =
```

The last definition assigns "LIBES" the null string. A macro that is never explicitly defined has the null string as its value.

A macro is invoked by preceding the macro name with a dollar sign; macro names longer than one character must be placed in parentheses. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations.

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical.

Macros are typically used as placeholders for values that may change from time to time. For example, the following makefile uses one macro for the names of object files to be linked and one for the names of the library.

```
OBJECTS = x.o y.o z.o
LIBES = -lln
prog: $(OBJECTS)
        cc $(OBJECTS) $(LIBES) -o prog
```

If this makefile is invoked with the command:

```
make
```

it will load the three object files with the **lex**(CP) library specified with the **-lln** option.

You may include a macro definition in a command line. A macro definition in a command line has the same form as a macro definition in a makefile. If spaces are to be used in the definition, double quotation marks must be used to enclose the definition. Macros in a command line override corresponding definitions found in the makefile.

For example, the command:

```
make "LIBES=-lln -lm"
```

loads, and assigns the library options **-lln** and **-lm** to "LIBES".

You can modify all or part of the value generated from a macro invocation without changing the macro itself by using the "substitution sequence". The sequence has the form

name : *st1* =[*st2*]

where *name* is the name of the macro whose value is to be modified, *st1* is the character or characters to be modified, and *st2* is the character or characters to replace the modified characters. If *st2* is not given, *st1* is replaced by a null character.

The substitution sequence is typically used to allow user-defined metacharacters in a makefile. For example, suppose that ".x" is to be used as a metacharacter for a prefix and suppose that a makefile contains the definition

```
FILES = prog1.x prog2.x prog3.x
```

Then the macro invocation

```
$(FILES : .x=.o)
```

generates the value

```
prog1.o prog2.o prog3.o
```

The actual value of "FILES" remains unchanged.

make has five built-in macros that can be used when writing dependency lines. The following is a list of these macros.

$*	Contains the name of the current target with the suffix removed. Thus, if the current target is *prog.o*, **$*** contains *prog*. It may be used in dependency lines that redefine the built-in rules.
$@	Contains the full pathname of the current target. It may be used in dependency lines with user-defined target names.
$<	Contains the filename of the dependent that is more recent than the given target. It may be used in dependency lines with built-in target names or the **.DEFAULT** pseudo-target name.

$?	Contains the filenames of the dependents that are more recent than the given target. It may be used in dependency lines with user-defined target names.
$%	Contains the filename of a library member. It may be used with target library names (see the section "Using Libraries" later in this chapter). In this case, **$@** contains the name of the library and **$%** contains the name of the library member.

You can change the meaning of a built-in macro by appending the **D** or **F** descriptor to its name. A built-in macro with the **D** descriptor contains the name of the directory containing the given file. If the file is in the current directory, the macro contains (.). A macro with the **F** descriptor contains the name of the given file with the directory name part removed. The **D** and **F** descriptor must not be used with the **$?** macro.

2.6 Using Shell Environment Variables

make provides access to current values of the shell's environment variables such as "HOME", "PATH", and "LOGIN". **make** automatically assigns the value of each shell variable in your environment to a macro of the same name. You can access a variable's value in the same way that you access the value of explicitly defined macros. For example, in the following dependency line, "$(HOME)" has the same value as the user's "HOME" variable.

```
prog:
        cc $(HOME)/x.o $(HOME)/y.o /usr/pub/z.o
```

make assigns the shell variable values after it assigns values to the built-in macros, but before it assigns values to user-specified macros. Thus, you can override the value of a shell variable by explicitly assigning a value to the corresponding macro. For example, the following macro definition causes **make** to ignore the current value of the "HOME" variable and use */usr/pub* instead.

```
HOME = /usr/pub
```

If a makefile contain macro definitions that override the current values of the shell variables, you can direct **make** to ignore these definitions by using the **-e** option.

make has two shell variables, "MAKE" and "MAKEFLAGS", that correspond to two special-purpose macros.

The "MAKE" macro provides a way to override the **-n** option and execute selected commands in a makefile. When "MAKE" is used in a command,

make will always execute that command, even if **-n** has been given in the invocation. The variable may be set to any value or command sequence.

The "MAKEFLAGS" macro contains one or more **make** options, and can be used in invocations of **make** from within a makefile. You may assign any **make** options to "MAKEFLAGS" except **-f**, **-p**, and **-d**. If you do not assign a value to the macro, **make** automatically assigns the current options to it, i.e., the options given in the current invocation.

The "MAKE" and "MAKEFLAGS" variables, together with the **-n** option, are typically used to debug makefiles that generate entire software systems. For example, in the following makefile, setting "MAKE" to "make" and invoking this file with the **-n** options displays all the commands used to generate the programs *prog1*, *prog2*, and *prog3* without actually executing them.

```
system : prog1 prog2 prog3
        @echo System complete.

prog1 : prog1.c
        $(MAKE) $(MAKEFLAGS) prog1

prog2 : prog2.c
        $(MAKE) $(MAKEFLAGS) prog2

prog3 : prog3.c
        $(MAKE) $(MAKEFLAGS) prog3
```

2.7 Using the Built-In Rules

make provides a set of built-in dependency lines, called built-in rules, that automatically check the targets and dependents given in a makefile, and create up-to-date versions of these files, if necessary. The built-in rules are identical to user-defined dependency lines except that they use the suffix of the filename as the target or dependent instead of the filename itself. For example, **make** automatically assumes that all files with the suffix *.o* have dependent files with the suffixes *.c* and *.s*.

When no explicit dependency line for a given file is given in a makefile, **make** automatically checks the default dependents of the file. It then forms the name of the dependents by removing the suffix of the given file and appending the predefined dependent suffixes. If the given file is out of date with respect to these default dependents, **make** searches for a built-in rule that defines how to create an up-to-date version of the file, then executes it. There are built-in rules for the following files.

.o	Object file
.c	C source file
.r	Ratfor source file
.f	Fortran source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.l	Lex source grammar

For example, if the file *x.o* is needed and there is an *x.c* in the description or directory, it is compiled. If there is also an *x.l*, that grammar would be run through **lex**(CP) before compiling the result.

The built-in rules are designed to reduce the size of your makefiles. They provide the rules for creating common files from typical dependents. Reconsider the example given in the section 2.2, "Creating a Makefile". In this example, the program *prog* depended on three object files *x.o*, *y.o*, and *z.o*. These files in turn depended on the C language source files *x.c*, *y.c*, and *z.c*. The files *x.c* and *y.c*, also depend on the include file, *defs*. In the original example, each dependency and corresponding command sequence was explicitly given. Many of these dependency lines were unnecessary, since the built-in rules could have been used instead. The following is all that is needed to show the relationships between these files:

```
prog: x.o y.o z.o
        cc x.o y.o z.o -o prog

x.o y.o: defs
```

In this makefile, *prog* depends on three object files, and an explicit command is given showing how to update *prog*. However, the second line merely shows that two objects files depend on the include file *defs*. No explicit command sequence is given on how to update them if necessary. Instead, **make** uses the built-in rules to locate the desired C source files, compile these files, and create the necessary object files.

2.8 Changing the Built-in Rules

You can change the built-in rules by redefining the macros used in these lines or by redefining the commands associated with the rules. You can display a complete list of the built-in rules and macros used in the rules by entering:

```
make -fp - 2>/dev/null </dev/null
```

The rules and macros are displayed on the standard output.

The macros of the built-in dependency lines define the names and options of the compilers, program generators, and other programs invoked by the built-in commands. **make** automatically assigns a default value to these macros when you start the program. You can change the values by redefining the macros in your makefile. For example, the following built-in rule contains three macros, "CC", "CFLAGS", and "LOADLIBES":

```
.c:
        $(CC) $(CFLAGS) $< $(LOADLIBES) -o $@
```

You can redefine any of these macros by placing the appropriate macro definition at the beginning of the makefile.

You can redefine the action of a built-in rule by giving a new rule in your makefile. A built-in rule has the following format:

```
suffix-rule:
        command
```

where *suffix-rule* is a combination of suffixes showing the relationship of the implied target and dependent, and *command* is the XENIX command required to carry out the rule. If more than one command is needed, they are given on separate lines.

The new rule must begin with an appropriate *suffix-rule*. The available *suffix-rules* are:

.c	.c~
.sh	.sh~
.c.o	.c~.o
.c~.c	.s.o
.s~.o	.y.o
.y~.o	.l.o
.l~.o	.y.c
.y~.c	.l.c
.c.a	.c~.a
.s~.a	.h~.h

A tilde (~) indicates an SCCS file. A single suffix indicate a rule that makes an executable file from the given file. For example, the suffix rule ".c" is for the built-in rule that creates an executable file, from a C source file. A pair of suffixes indicates a rule that makes one file from the other. For example, ".c.o" is for the rule that creates an object file (*.o*) file from a corresponding C source file (*.c*).

Any commands in the rule may use the built-in macros provided by **make**. For example, the following dependency line redefines the action of the *.c.o* rule:

```
.c.o :
        cc68 $< -c $*.o
```

If necessary, you can also create new *suffix-rules* by adding a list of new suffixes to a makefile with ".SUFFIXES". This pseudo-target name defines the suffixes that may be used to make *suffix-rules* for the built-in rules. The line has the form:

.SUFFIXES: *suffix* ...

where *suffix* is a lowercase letter preceded by a dot (.). If more than one suffix is given, you must use spaces to separate them.

The order of the suffixes is significant. Each suffix is a dependent of the suffixes preceding it. For example, the suffix list:

```
.SUFFIXES: .o .c .y .l .s
```

causes *prog.c* to be a dependent of *prog.o*, and *prog.y* to be a dependent of *prog.c*.

You can create new *suffix-rules* by combining dependent suffixes with the suffix of the intended target. The dependent suffix must appear first.

If a ".SUFFIXES" list appears more than once in a makefile, the suffixes are combined into a single list. If a ".SUFFIXES" is given without a list, all suffixes are ignored.

2.9 Using Libraries

You can direct **make** to use a file contained in an archive library as a target or dependent. To do this, you must explicitly name the file you wish to access by using a library name. A library name has the form:

lib(*member-name*)

where *lib* is the name of the library containing the file, and *member-name* is the name of the file. For example, the library name:

```
libtemp.a(print.o)
```

refers to the object file *print.o*, in the archive library *libtemp.a*.

You can create your own built-in rules for archive libraries by adding the *.a* suffix to the suffix list, and creating new suffix combinations. For example, the combination ".c.a" may be used for a rule that defines how to create a library member from a C source file. Note that the dependent suffix in the new combination must be different than the suffix of the ultimate file. For example, the combination ".c.a" can be used for a rule that creates *.o* files, but not for one that creates *.c* files.

The most common use of the library naming convention is to create a makefile that automatically maintains an archive library. For example, the following dependency lines define the commands required to create a library, named *lib*, that contains up to date versions of the files *file1.o*, *file2.o*, and *file3.o*.

```
lib:
        lib(file1.o) lib(file2.o) lib(file3.o)
        @echo lib is now up to date
.c.a:
        $(CC) -c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
```

The *.c.a* rule shows how to redefine a built-in rule for a library. In the following example, the built-in rule is disabled, allowing the first dependency to create the library.

```
lib:
        lib(file1.o) lib(file2.o) lib(file3.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        ar rv lib $?
        rm $?
        @echo lib is now up to date
.c.a:;
```

In this example, a substitution sequence is used to change the value of the "$?" macro from the names of the object files "file1.o", "file2.o", and "file3.o" to "file1.c", "file2.c", and "file3.c".

2.10 Troubleshooting

Most difficulties in using **make** arise from **make**'s specific meaning of dependency. If the file *x.c* has the following line:

```
#include "defs"
```

then the object file *x.o* depends on *defs*; the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, but it is necessary to recreate *x.o*.)

Use the **-n** option to get a listing of which commands **make** will execute, without actually executing them. For example, the command:

```
make -n
```

prints out a listing of the commands **make** would normally execute.

The debugging option, **-d**, causes **make** to print out a very detailed description of what it is doing, including the file times. The output is verbose, and recommended only as a last resort.

If a change to a file is small (for example, adding a new definition to an include file), the **-t** (touch) option can save a lot of time. Instead of constantly recompiling, **make** updates the modification times on the affected file. Thus, the command:

```
make -ts
```

which stands for touch silently, causes the relevant files to appear up to date.

2.11 Using make: An Example

Figure 2-1 gives an example of a makefile, used to maintain the **make** itself. The code for **make** is spread over a number of C source files and a *yacc* grammar.

make usually prints out each command before issuing it. The output shown below results from entering the following command:

```
make
```

in a directory containing only the source and makefile:

```
cc -c vers.c
cc -c main.c
cc -c doname.c
cc -c misc.c
cc -c files.c
cc -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -c gram.c
cc vers.o main.o ... dosys.o gram.o -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the makefile, **make** found them by using its suffix rules and issued the needed commands. The string of digits results from the **size make** command.

The last few targets in the makefile are useful maintenance sequences. The *print* target prints only the files that have been changed since the last **make print** command. A zero-length file, *print,* is maintained to keep track of the time of printing; the **$?** macro, in the command line, picks up only the names of the files changed since *print* was touched. The printed output can then be sent to a different printer, or to a file, by changing the definition of the **P** macro.

Figure 2-1. Makefile Contents

```
# Description file for the make command

# Macro definitions below
P = lpr
FILES = Makefile vers.c defs main.c doname.c misc.c files.c dosys.c\
        gram.y lex.c
OBJECTS = vers.o main.o ... dosys.o gram.o
LIBES=
LINT = lint -p
CFLAGS = -O

#targets: dependents
#<TAB> actions

make: $(OBJECTS)
        cc $(CFLAGS)  $(OBJECTS)  $(LIBES)  -o make
        size make

$(OBJECTS): defs
gram.o: lex.c

cleanup:
        -rm *.o gram.c
        -du

install:
        @size make  /usr/bin/make
        cp make /usr/bin/make ;  rm make

print: $(FILES)        # print recently changed files
        pr $? |$P
        touch print

test:
        make -dp |grep  -v TIME > 1zap
        /usr/bin/make  -dp |grep -v TIME > 2zap
        diff 1zap 2zap
        rm 1zap 2zap

lint : dosys.c doname.c files.c main.c misc.c vers.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c vers.c gram.c
        rm gram.c

arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

Chapter 3

SCCS: A Source Code Control System

3.1 Introduction 3-1

3.2 Basic Information 3-1
- 3.2.1 Files and Directories 3-1
- 3.2.2 Deltas and SIDs 3-2
- 3.2.3 SCCS Working Files 3-3
- 3.2.4 SCCS Command Arguments 3-4
- 3.2.5 File Administrator 3-5

3.3 Creating and Using S-files 3-5
- 3.3.1 Creating an S-file 3-5
- 3.3.2 Retrieving a File for Reading 3-6
- 3.3.3 Retrieving a File for Editing 3-7
- 3.3.4 Saving a New Version of a File 3-8
- 3.3.5 Retrieving a Specific Version 3-9
- 3.3.6 Changing the Release Number of a File 3-10
- 3.3.7 Creating a Branch Version 3-11
- 3.3.8 Retrieving a Branch Version 3-11
- 3.3.9 Retrieving the Most Recent Version 3-12
- 3.3.10 Displaying a Version 3-12
- 3.3.11 Saving a Copy of a New Version 3-13
- 3.3.12 Displaying Helpful Information 3-13

3.4 Using Identification Keywords 3-14
- 3.4.1 Inserting a Keyword into a File 3-14
- 3.4.2 Assigning Values to Keywords 3-15
- 3.4.3 Forcing Keywords 3-15

3.5 Using S-file Flags 3-16
- 3.5.1 Setting S-file Flags 3-16
- 3.5.2 Using the i Flag 3-16
- 3.5.3 Using the d Flag 3-17
- 3.5.4 Using the v Flag 3-17
- 3.5.5 Removing an S-file Flag 3-17

3.6 Modifying S-file Information 3-18

3.6.1 Adding Comments 3-18
3.6.2 Changing Comments 3-18
3.6.3 Adding Modification Requests 3-19
3.6.4 Changing Modification Requests 3-20
3.6.5 Adding Descriptive Text 3-20

3.7 Printing from an S-file 3-21
3.7.1 Using a Data Specification 3-21
3.7.2 Printing a Specific Version 3-22
3.7.3 Printing Later and Earlier Versions 3-22

3.8 Editing by Several Users 3-22
3.8.1 Editing Different Versions 3-23
3.8.2 Editing a Single Version 3-23
3.8.3 Saving a Specific Version 3-23

3.9 Protecting S-files 3-24
3.9.1 Adding a User to the User List 3-24
3.9.2 Removing a User from a User List 3-25
3.9.3 Setting the Floor Flag 3-25
3.9.4 Setting the Ceiling Flag 3-25
3.9.5 Locking a Version 3-25

3.10 Repairing SCCS Files 3-26
3.10.1 Checking an S-file 3-26
3.10.2 Editing an S-file 3-27
3.10.3 Changing an S-file's Checksum 3-27
3.10.4 Regenerating a G-file for Editing 3-27
3.10.5 Restoring a Damaged P-file 3-27

3.11 Using Other Command Options 3-28
3.11.1 Getting Help With SCCS Commands 3-28
3.11.2 Creating a File With the Standard Input 3-28
3.11.3 Starting At a Specific Release 3-28
3.11.4 Adding a Comment to the First Version 3-29
3.11.5 Suppressing Normal Output 3-29
3.11.6 Including and Excluding Deltas 3-30
3.11.7 Listing the Deltas of a Version 3-31
3.11.8 Mapping Lines to Deltas 3-31
3.11.9 Naming Lines 3-31
3.11.10 Displaying a List of Differences 3-32
3.11.11 Displaying File Information 3-32
3.11.12 Removing a Delta 3-32
3.11.13 Searching for Strings 3-33
3.11.14 Comparing SCCS Files 3-33

3.1 Introduction

The Source Code Control System, (SCCS) is a collection of XENIX commands that create, maintain, and control special files called SCCS files. The SCCS commands let you create and store multiple versions of a program or document in a single file, instead of having one file for each version. The commands let you retrieve any version you wish at any time, make changes to this version, and save the changes as a new version of the file, in the SCCS file.

The SCCS system is useful wherever you require a compact way to store multiple versions of the same file. The SCCS system provides an easy way to update any given version of a file and explicitly record the changes made. The commands are typically used to control changes to multiple versions of source programs, but may also be used to control multiple versions of manuals, specifications, and other documentation.

This chapter explains how to make SCCS files, how to update the files contained in SCCS files, and how to maintain the SCCS files once they are created. The following sections describe the basic information you need to start using the SCCS commands. Later sections describe the commands in detail.

3.2 Basic Information

This section provides some basic information about the SCCS system. In particular, it describes:

- Files and directories
- Deltas and SIDs
- SCCS working files
- SCCS command arguments
- File administration

3.2.1 Files and Directories

All SCCS files (also called s-files) are originally created from text files containing documents or programs created by a user. The text files must have been created using a XENIX text editor such as **vi**(C). Special characters in the files are allowed only if they are also allowed by the given editor.

To simplify s-file storage, all logically related files (e.g., files belonging to the same project) should be kept in the same directory. Such directories should contain s-files only, and should have read and examine permission for everyone, and write permission for the user only.

Note that you must not use the XENIX **ln**(C) command to create multiple copies of an s-file.

3.2.2 Deltas and SIDs

Unlike an ordinary text file, an SCCS file (or s-file for short) contains nothing more than lists of changes. Each list corresponds to the changes needed to construct exactly one version of the file. The lists can then be combined to create the desired version from the original.

Each list of changes is called a "delta". Each delta has an identification string called an "SID". The SID is a string of at least two, and at most four, numbers separated by periods. The numbers name the version and define how it is related to other versions. For example, the first delta is usually numbered 1.1 and the second 1.2.

The first number in any SID is called the "release number". The release number usually indicates a group of versions that are similar and generally compatible. The second number in the SID is the "level number". It indicates major differences between files in the same release.

An SID may also have two optional numbers. The "branch number", known as the optional third number, indicates changes at a particular level, and the "sequence number", the fourth number, indicates changes at a particular branch. For example, the SIDs 1.1.1.1 and 1.1.1.2 indicate two new versions that contain slight changes to the original delta 1.1.

An s-file may at any time contain several different releases, levels, branches, and sequences of the same file. In general, the maximum number of releases an s-file may contain is 9999, that is, release numbers may range from 1 to 9999. The same limit applies to level, branch, and sequence numbers.

When you create a new version, the SCCS system usually creates a new SID by incrementing the level number of the original version. If you wish to create a new release, you must explicitly instruct the system to do so. A change to a release number indicates a major new version of the file. How to create a new version of a file, and change release numbers, are described later.

The SCCS system creates a branch and sequence number for the SID of a new version, only if the next higher level number already exists. For example, if you change version 1.3 to create a version 1.4 and then change 1.3 again, the SCCS system creates a new version named 1.3.1.1.

Version numbers can become quite complicated. In general, it is wise to keep the numbers as simple as possible by carefully planning the creation of each new version.

3.2.3 SCCS Working Files

The SCCS system uses several different kinds of files to complete its tasks. In general, these files contain either actual text, or information about the commands in progress. For convenience, the SCCS system names these files by placing a prefix before the name of the original file from which all versions were made. The following is a list of the working files:

s-file	A permanent file that contains all versions of the given text file. The versions are stored as deltas, that is, lists of changes to be applied to the original file to create the given version. The name of an s-file is formed by placing the file prefix *s.* at the beginning of the original filename.
x-file	A temporary copy of the s-file. It is created by SCCS commands which change the s-file. It is used instead of the s-file to carry out the changes. When all changes are complete, the SCCS system removes the original s-file and gives the x-file the name of the original s-file. The name of the x-file is formed by placing the prefix *x.* at the beginning of the original file.
g-file	An ordinary text file created by applying the deltas in a given s-file to the original file. The g-file represents a copy of the given version of the original file, and receives the same filename as the original. When created, a g-file is placed in the current working directory of the user who requested the file.
p-file	A special file containing information about the versions of an s-file currently being edited. The p-file is created when a g-file is retrieved from the s-file. The p-file exists until all currently retrieved files have been saved in the s-file; it is then deleted. The p-file contains one or more entries describing the SID of the retrieved g-file, the proposed SID of the new, edited g-file, and the login name of the user who retrieved the g-file. The p-file name is formed by placing the prefix *p.* at the beginning of the original filename.
z-file	A lock file is used by SCCS commands to prevent two users from updating a single SCCS file at the same time. Before a command modifes an SCCS file, it creates a z-file and copies its own process ID to it. Any other command which attempts to access the file while the z-file is present displays an error message and stops. When the original command

	has finished its tasks, it deletes the z-file before stopping. The z-file name is formed by placing the prefix *z.* at the beginning of the original filename.
l-file	A special file containing a list of the deltas required to create a given version of a file. The l-file name is formed by placing the prefix *l.* at the beginning of the original filename.
d-file	A temporary copy of the g-file used to generate a new delta.
q-file	A temporary file used by the **delta**(CP) command when updating the p-file. The file is not directly accessible.

In general, a user never directly accesses x-files, z-files, d-files, or q-files. If a system crash or similar situation abnormally terminates a command, the user may wish to delete these files to ensure proper operation of subsequent SCCS commands.

3.2.4 SCCS Command Arguments

Almost all SCCS commands accept two types of arguments: options and filenames. These appear in the SCCS command line immediately after the command name.

An option indicates a special action to be taken by the given SCCS command. An option is usually a lowercase letter preceded by a minus sign (-). Some options require an additional name or value.

A filename indicates the file to be acted on. The syntax for SCCS filenames is like other XENIX filename syntax. Appropriate pathnames must be given if required. Some commands also allow directory names. In this case, all files in the directory are acted on. If the directory contains non-SCCS and unreadable files, these are ignored. A filename must not begin with a minus sign (-).

The special symbol - may be used to cause the given command to read a list of filenames from the standard input. These filenames are then used as names for the files to be processed. The list must terminate with an end-of-file character.

Any options given with a command apply to all files. The SCCS commands process the options before any filenames, so the options may appear anywhere on the command line.

Filenames are processed left to right. If a command encounters a fatal error, it stops processing the current file and, if any other files have been given, begins processing the next.

3.2.5 File Administrator

Every SCCS file requires an administrator to maintain and keep the file in order. The administrator is usually the user who created the file and therefore owns it. Before other users can access the file, the administrator must ensure that they have adequate access. Several SCCS commands let the administrator define who has access to the versions in a given s-file. These are described later.

3.3 Creating and Using S-files

The s-file is the key element in the SCCS system. It provides compact storage for all versions of a given file and automatic maintenance of the relationships between the versions.

This section explains how to use the **admin**(CP), **get**(CP), and **delta**(CP) commands to create and use s-files. In particular, it describes how to create the first version of a file, how to retrieve versions for reading and editing, and how to save new versions.

3.3.1 Creating an S-file

You can create an s-file from an existing text file using the **-i** (for "initialize") option of the **admin** command. The command has the form:

admin -i*filename s.filename*

where -i*filename* gives the name of the text file from which the s-file is to be created, and *s.filename* is the name of the new s-file. The name must begin with *s.* and must be unique; no other s-file in the same directory may have the same name. For example, suppose the file named *demo.c* contains the following C language program:

```
#include <stdio.h>

main ()
{
        printf("This is version 1.1 \n");
}
```

To create an s-file, enter:

```
admin -idemo.c s.demo.c
```

This command creates the s-file *s.demo.c*, and copies the first delta describing the contents of *demo.c* to this new file. The first delta is numbered 1.1.

After creating an s-file, the original text file should be removed using the **rm** command, since it is no longer needed. If you wish to view the text file or make changes to it, you can retrieve the file using the **get** command described in the next section.

When first creating an s-file, the **admin** command may display the warning message:

```
No id keywords (cm7)
```

In general, this message can be ignored unless you have specifically included keywords in your file (see the section, "Using Identification Keywords" later in this chapter).

Note that only a user with write permission in the directory containing the s-file may use the **admin** command on that file. This protects the file from administration by unauthorized users.

3.3.2 Retrieving a File for Reading

You can retrieve a file for reading from a given s-file by using the **get** command. The command has the form:

get *s.filename ...*

where *s.filename* is the name of the s-file containing the text file. The command retrieves the lastest version of the text file and copies it to a regular file. The file has the same name as the s-file but with the *s.* removed. It also has read-only file permissions. For example, suppose the s-file, *s.demo.c*, contains the first version of the short C program shown in the previous section. To retrieve this program, enter:

```
get s.demo.c
```

The command retrieves the program and copies it to the file named *demo.c*. You may then display the file just as you do any other text file.

The command also displays a message which describes the SID of the retrieved file and its size in lines. For example, after retrieving the short C program from *s.demo.c*,
the command displays the message:

```
1.1
6 lines
```

You may also retrieve more than one file at a time by giving multiple s-file names in the command line. For example, the command:

```
get s.demo.c s.def.h
```

retrieves the contents of the s-files *s.demo.c* and *s.def.h* and copies them to the text files *demo.c* and *def.h*. When giving multiple s-file names in a command, you must separate each with at least one space. When the **get** command displays information about the files, it places the corresponding filename before the relavent information.

3.3.3 Retrieving a File for Editing

You can retrieve a file for editing from a given s-file by using the **-e** (for "editing") option of the **get** command. The command has the form:

get -e *s.filename* ...

where *s.filename* is the name of the s-file containing the text file. You may give more than one filename if you wish. If you do, you must separate each name with a space.

The command retrieves the lastest version of the text file and copies it to an ordinary text file. The file has the same name as the s-file but with the *s.* removed. It has read and write file permissions. For example, suppose the s-file, *s.demo.c*, contains the first version of a C program. To retrieve this program, enter:

```
get -e s.demo.c
```

The command retrieves the program and copies it to the file named *demo.c*. You may edit the file just as you do any other text file.

If you give more than one filename, the command creates files for each corresponding s-file. Since the **-e** option applies to all the files, you may edit each one.

After retrieving a text file, the command displays a message giving the SID of the file and its size in lines. The message also displays a proposed SID, that is, the SID for the new version after editing. For example, after retrieving the six-line C program in *s.demo.c*, the command displays the following message:

```
1.1
new delta 1.2
6 lines
```

The proposed SID is 1.2. If more than one file is retrieved, the corresponding filename precedes the relevant information.

Note that any changes made to the text file are not immediately copied to the corresponding s-file. To save these changes, you must use the **delta** command described in the next section. To help keep track of the current file version, the **get** command creates another file, called a p-file, that contains information about the text file. This file is used by a subsequent **delta** command when saving the new version. The p-file has the same name as the s-file but begins with a *p.*. The user must not access the p-file directly.

3.3.4 Saving a New Version of a File

You can save a new version of a text file by using the **delta** command. The command has the form:

```
delta s.filename
```

where *s.filename* is the name of the s-file from which the modified text file was retrieved. For example, to save changes made to a C program in the file *demo.c* (which was retrieved from the file *s.demo.c*), enter:

```
delta s.demo.c
```

Before saving the new version, the **delta** command asks for comments explaining the nature of the changes. It displays the following prompt:

```
comments?
```

You may type any text you think appropriate, up to 512 characters. The comment must end with a newline character. If necessary, you can start a new line by typing a backslash (\) followed by a newline character. If you do not wish to include a comment, just type a newline character.

Once you have given a comment, the command uses the information in the corresponding p-file to compare the original version with the new version. A list of all the changes is copied to the s-file. This is the new delta.

After a command has copied the new delta to the s-file, it displays a message showing the new SID and the number of lines inserted, deleted, or left unchanged in the new version. For example, if the C program has been changed to:

```
#include <stdio.h>

main ()
{
        int i=2;

        printf("This is version 1.%d0, i);
}
```

the command displays the message:

```
1.2
3 inserted
1 deleted
5 unchanged
```

Once a new version is saved, the next **get** command retrieves the new version. The command ignores previous versions. If you wish to retrieve a previous version, you must use the **-r** option of the **get** command, as described in the next section.

3.3.5 Retrieving a Specific Version

You can retrieve any version you wish from an s-file by using the **-r** (for "retrieve") of the **get** command. The command has the form:

get [-e] -r*SID s.filename ...*

where **-e** is the edit option, -r*SID* gives the SID of the version to be retrieved, and *s.filename* is the name of the s-file containing the file to be retrieved. You may give more than one filename. The names must be separated with spaces.

The command retrieves the given version, and copies it to the file having the same name as s-file, but with the *s.* removed. The file has read-only permission unless you also give the **-e** option. If multiple filenames are given, one text file of the given version is retrieved from each. For example, the command:

```
get -r1.1 s.demo.c
```

retrieves version 1.1 from the s-file *s.demo.c*, but the command:

```
get -e -r1.1 s.demo.c s.def.h
```

retrieves for editing a version 1.1 from both *s.demo.c* and *s.def.h*. If you give the number of a version that does not exist, the command displays an error message.

You may omit the level number of a version number if you wish, by just giving the release number. If you do, the command automatically retrieves the most recent version having the same release number. For example, if the most recent version in the file *s.demo.c* is numbered 1.4, the command:

```
get -r1 s.demo.c
```

retrieves version 1.4. If there is no version with the given release number, the command retrieves the most recent version in the previous release.

3.3.6 Changing the Release Number of a File

You can direct the **delta** command to change the release number of a new version of a file by using the **-r** option of the **get** command. In this case, the **get** command has the form:

get -e -r*rel-num s.filename* ...

where **-e** is the required edit option, -r*rel-num* is the new release number of the file, and *s.filename* is the name of the s-file containing the file to be retrieved. The new release number must be an entirely new number, that is, no existing version may have this number. You may give more than one filename.

The command retrieves the most recent version from the s-file, then copies the new release number to the p-file. On the subsequent **delta** command, the new version is saved using the new release number and level number 1. For example, if the most recent version in the s-file *s.demo.c* is 1.4, the command:

```
get -e -r2 s.demo.c
```

causes the subsequent **delta** to save a new version 2.1, not 1.5. The new release number applies to the new version only; the release numbers of previous versions are not affected. Therefore, if you edit version 1.4 (from which 2.1 was derived) and save the changes, you create a new version 1.5. Similarly, if you edit version 2.1, you create a new version 2.2.

As before, the **get** command also displays a message showing the current version number, the proposed version number, and the size of the file in lines. Similarly, the subsequent **delta** command displays the new version number and the number of lines inserted, deleted, and unchanged in the new file.

3.3.7 Creating a Branch Version

You can create a branch version of a file by editing a version that has been previously edited. A branch version is simply a version whose SID contains a branch and sequence number.

For example, if version 1.4 already exists, the command:

```
get -e -r1.3 s.demo.c
```

retrieves version 1.3 for editing and gives 1.3.1.1 as the proposed SID.

In general, whenever **get** discovers that you wish to edit a version that already has a succeeding version, it uses the first available branch and sequence numbers for the proposed SID. For example, if you edit version 1.3 a third time, **get** gives 1.3.2.1 as the proposed SID.

You can save a branch version just like any other version by using the **delta** command.

3.3.8 Retrieving a Branch Version

You can retrieve a branch version of a file by using the **-r** option of the **get** command.
For example, the command:

```
get -r1.3.1.1 s.demo.c
```

retrieves branch version 1.3.1.1.

You may retrieve a branch version for editing by using the **-e** option of the **get** command. When retrieving for editing, **get** creates the proposed SID by incrementing the sequence number by one. For example, if you retrieve branch version 1.3.1.1 for editing, **get** gives 1.3.1.2 as the proposed SID.

As always, the command displays the version number and file size. If the given branch version does not exist, the command displays an error message.

You may omit the sequence number if you wish. In this case, the command retrieves the most recent branch version with the given branch number.

For example, if the most recent branch version in the s-file *s.def.h* is 1.3.1.4, the command:

```
get -r1.3.1 s.def.h
```

retrieves version 1.3.1.4.

3.3.9 Retrieving the Most Recent Version

You can always retrieve the most recent version of a file by using the **-t** option with the **get** command. For example, the command:

```
get -t s.demo.c
```

retrieves the most recent version from the file *s.demo.c*. You may combine the **-r** and **-t** options, to retrieve the most recent version of a given release number. For example, if the most recent version with release number 3 is 3.5, then the command:

```
get -r3 -t s.demo.c
```

retrieves version 3.5. If a branch version exists that is more recent than version 3.5 (e.g., 3.2.1.5), then the above command retrieves the branch version and ignores version 3.5.

3.3.10 Displaying a Version

You can display the contents of a version on the standard output by using the **-p** option of the **get** command. For example, the command:

```
get -p s.demo.c
```

displays the most recent version in the s-file, *s.demo.c*, on the standard output. Similarly, the command:

```
get -p -r2.1 s.demo.c
```

displays version 2.1 on the standard output.

The **-p** option is useful for creating g-files with user-supplied names. This option also directs all output normally sent to the standard output, such as the SID of the retrieved file, to the standard error file. Thus, the resulting file contains only the contents of the given version. For example, the command:

```
get -p s.demo.c >version.c
```

copies the most recent version in the s-file, *s.demo.c,* to the file *version.c.* The SID of the file and its size is copied to the standard error file.

3.3.11 Saving a Copy of a New Version

The **delta** command normally removes the edited file after saving it in the s-file. You can save a copy of this file by using the **-n** option of the **delta** command. For example, the command:

```
delta -n s.demo.c
```

first saves a new version in the s-file, *s.demo.c,* then saves a copy of this version in the file *demo.c.* You may display the file as desired, but you cannot edit the file.

3.3.12 Displaying Helpful Information

An SCCS command displays an error message whenever it encounters an error in a file. An error message has the form:

ERROR [*filename*]: *message* (*code*)

where *filename* is the name of the file being processed, *message* is a short description of the error, and *code* is the error code.

You may use the error code as an argument to the **help** command to display additional information about the error. The command has the form:

help *code*

where *code* is the error code given in an error message. The command displays one or more lines of text that explain the error, and suggests a possible remedy. For example, the command:

```
help co1
```

displays the message

```
col:
"not a SCCS file"
A file that you think is a SCCS file
does not begin with the characters "s.".
```

The **help** command can be used at any time.

3.4 Using Identification Keywords

The SCCS system provides several special symbols, called identification keywords, which may be used in the text of a program or document to represent a predefined value. Keywords represent a wide range of values, from the creation date and time of a given file, to the name of the module containing the keyword. When a user retrieves the file for reading, the SCCS system automatically replaces any keywords it finds in a given version of a file with the keyword's value.

This section explains how keywords are treated by the various SCCS commands, and how you may use the keywords in your own files. Only a few keywords are described in this section. For a complete list of the keywords, see the section **get**(CP) in the XENIX *Reference Manual.*

3.4.1 Inserting a Keyword into a File

You may insert a keyword into any text file. A keyword is simply an uppercase letter enclosed in percent signs (%). No special characters are required. For example, "%I%" is the keyword representing the SID of the current version, and "%H%" is the keyword representing the current date.

When the program is retrieved for reading using the **get** command, the keywords are replaced by their current values. For example, if the "%M%", "%I%", and "%H" keywords are used in place of the module name, SID, and current data in the following program statement:

```
char header(100) = {"%M% %I% %H% "};
```

then these keywords are expanded in the retrieved version of the program as shown below:

```
char header(100) = {"MODNAME 2.3 07/07/77 "};
```

The **get** command does not replace keywords when retrieving a version for editing. The system assumes that you wish to keep the keywords (and not their values) when you save the new version of the file.

To indicate that a file has no keywords, the **get, delta**, and **admin** commands display the message:

```
No id keywords (cm7)
```

This message is normally treated as a warning, letting you know that no keywords are present. However, you may change the operation of the system to make this a fatal error, as explained later in this chapter.

3.4.2 Assigning Values to Keywords

The values of most keywords are predefined by the system, but some, such as the value for the "%M%" keyword, can be explicitly defined by the user. To assign a value to a keyword, you must set the corresponding s-file flag to the desired value. You can do this by using the **-f** option of the **admin** command.

For example, to set the **%M%** keyword to "cdemo", you must set the **m** flag as shown in the following command:

```
admin -fmcdemo s.demo.c
```

This command records "cdemo" as the current value of the **%M%** keyword. Note that if you do not set the **m** flag, the SCCS system uses the name of the original text file for **%M%**, by default.

The **t** and **q** flags are also associated with keywords. A description of these flags and the corresponding keywords can be found in the section **get**(CP) in the XENIX *Reference Manual.* You can change keyword values at any time.

3.4.3 Forcing Keywords

If a version is found to contain no keywords, you can force a fatal error by setting the **i** flag in the given s-file. The flag causes the **delta** and **admin** commands to stop processing of the given version and report an error. The flag is useful for ensuring that keywords are used properly in a given file.

To set the **i** flag, you must use the **-f** option of the **admin** command. For example, the command:

```
admin -fi s.demo.c
```

sets the **i** flag in the s-file, *s.demo.c*. If the given version does not contain keywords, subsequent **delta** or **admin** commands that access this file print an error message.

Note that if you attempt to set the **i** flag at the same time that you create an s-file, and if the initial text file contains no keywords, the **admin** command displays a fatal error message, and stops without creating the s-file.

3.5 Using S-file Flags

An s-file flag is a special value that defines how a given SCCS command will operate on the corresponding s-file. The s-file flags are stored in the s-file and are read by each SCCS command before it operates on the file. S-file flags affect operations such as keyword checking, keyword replacement values, and default values for commands.

This section explains how to set and use s-file flags. It also describes the action of commonly-used flags. For a complete description of all flags, see the section **admin**(CP) in the XENIX *Reference Manual.*

3.5.1 Setting S-file Flags

You can set the flags in a given s-file by using the **-f** option of the **admin** command. The command has the form:

```
admin -fflag s.filename
```

where -f*flag* gives the flag to be set, and *s.filename* gives the name of the s-file in which the flag is to be set. For example, the command:

```
admin -fi s.demo.c
```

sets the **i** flag in the s-file *s.demo.c.*

Note that some s-file flags take values when they are set. For example, the **m** flag requires that a module name be given. When a value is required, it must immediately follow the flag name, as in the command:

```
admin -fmdmod s.demo.c
```

which sets the **m** flag to the module name "dmod".

3.5.2 Using the i Flag

The **i** flag causes the **admin** and **delta** commands to print a fatal error message and stop, if no keywords are found in the given text file. The flag is used to prevent a version of a file, which contains expanded keywords, from being saved as a new version. (Saving an expanded version destroys the keywords for all subsequent versions).

When the **i** flag is set, each new version of a file must contain at least one keyword. Otherwise, the version cannot be saved.

3.5.3 Using the d Flag

The **d** flag gives the default SID for versions retrieved by the **get** command. The flag takes an SID as its value. For example, the command:

```
admin -fd1.1 s.demo.c
```

sets the default SID to 1.1. A subsequent **get** command which does not use the **-r** option will retrieve version 1.1.

3.5.4 Using the v Flag

The **v** flag allows you to include modification requests in an s-file. Modification requests are names or numbers that may be used as a shorthand means of indicating the reason for each new version.

When the **v** flag is set, the **delta** command asks for the modification requests just before asking for comments. The **v** flag also allows the **-m** option to be used in the **delta** and **admin** commands.

3.5.5 Removing an S-file Flag

You can remove an s-file flag from an s-file by using the **-d** option of the **admin** command. The command has the form:

admin -d*flag s.filename*

where -d*flag* gives the name of the flag to be removed and *s.filename* is the name of the s-file from which the flag is to be removed. For example, the command:

```
admin -di s.demo.c
```

removes the **i** flag from the s-file *s.demo.c*. When removing a flag which takes a value, only the flag name is required. For example, the command:

```
admin -dm s.demo.c
```

removes the **m** flag from the s-file.

The **-d** and **-i** options must not be used at the same time.

3.6 Modifying S-file Information

Every s-file contains information about the deltas it contains. Normally, this information is maintained by the SCCS commands and is not directly accessible by the user. Some information, however, is specific to the user who creates the s-file, and may be changed as desired to meet the user's requirements. This information is kept in two special parts of the s-file called the "delta table" and the "description field".

The delta table contains information about each delta, such as the SID and the date and time of creation. It also contains user-supplied information, such as comments and modification requests. The description field contains a user-supplied description of the s-file and its contents. Both parts can be changed or deleted at any time to reflect changes to the s-file contents.

3.6.1 Adding Comments

You can add comments to an s-file by using the **-y** option of the **delta** and **admin** commands. This option causes the given text to be copied to the s-file as the comment for the new version. The comment may be any combination of letters, digits, and punctuation symbols. No embedded newline characters are allowed. If spaces are used, the comment must be enclosed in double quotes. The complete command must fit on one line. For example, the command:

```
delta -y"George Wheeler" s.demo.c
```

saves the comment "George Wheeler" in the s-file *s.demo.c.*

The **-y** option is typically used in shell procedures as part of an automated approach to maintaining files. When the option is used, the **delta** command does not print the corresponding comment prompt, so no interaction is required. If more than one s-file is given in the command line, the given comment applies to them all.

3.6.2 Changing Comments

You can change the comments in a given s-file by using the **cdc** command. The command has the form:

cdc -r*SID s.filename*

where -r*SID* gives the SID of the version whose comment is to be changed, and *s.filename* is the name of the s-file containing the version. The command asks for a new comment by displaying the prompt:

```
comments?
```

You may enter any sequence of characters up to 512 characters long. The sequence may contain embedded newline characters if they are preceded by a backslash (\). The sequence must be terminated with a newline character. For example, the command:

```
cdc -r3.4 s.demo.c
```

prompts for a new comment for version 3.4.

Although the command does not delete the old comment, it is no longer directly accessible by the user. The new comment contains the login name of the user who invoked the **cdc** command, and the time the comment was changed.

3.6.3 Adding Modification Requests

You can add modification requests to an s-file, when the **v** flag is set, by using the **-m** option of the **delta** and **admin** commands. A modification request is a shorthand method of describing the reason for a particular version. Modification requests are usually names or numbers which the user has chosen to represent a specific request.

The **-m** option causes the given command to save the requests following the option. A request may be any combination of letters, digits, and punctuation symbols. If you give more than one request, you must separate them with spaces and enclose the request in double quotes. For example, the command:

```
delta -m"error35 optimize10" s.demo.c
```

copies the requests "error35" and "optimize10" to *s.demo.c*, while saving the new version.

The **-m** option, when used with the **admin** command, must be combined with the **-i** option. Furthermore, the v flag must be explicitly set with the **-f** option. For example, the command:

```
admin -idef.h -m"error0" -fv s.def.h
```

inserts the modification request "error0," in the new file *s.def.h*.

The **delta** command does not prompt for modification requests if you use the **-m** option.

3.6.4 Changing Modification Requests

You can change modification requests, when the **v** flag is set, by using the **cdc** command. The command asks for a list of modification requests by displaying the prompt:

```
MRs?
```

You may enter any number of requests. Each request may have any combination of letters, digits, or punctuation symbols. No more than 512 characters are allowed, and the last request must be terminated with a newline character. If you wish to remove a request, you must precede the request with an exclamation mark (!). For example, the command:

```
cdc -r1.4 s.demo.c
```

asks for changes to the modification requests. By responding

```
MRs? error36 !error35
```

the request "error36" is added and "error35" is removed.

3.6.5 Adding Descriptive Text

You can add descriptive text to an s-file by using the **-t** option of the **admin** command. Descriptive text is any text that describes the purpose and reason for the given s-file. Descriptive text is independent of the contents of the s-file, and can only be displayed using the **prs** command.

The **-t** option directs the **admin** to copy the contents of a given file into the description field of the s-file. The command has the form:

admin -t*filename s.filename*

where -t*filename* gives the name of the file containing the descriptive text, and *s.filename* is the name of the s-file to receive the descriptive text. The file to be inserted may contain any amount of text. For example, the command:

```
admin -tcdemo s.demo.c
```

inserts the contents of the file *cdemo* into the description field of the s-file *s.demo.c.*

The **-t** option may also be used to initialize the description field when creating the s-file. For example, the command:

```
admin -idemo.c -tcdemo s.demo.c
```

inserts the contents of the file *cdemo* into the new s-file *s.demo.c*. If **-t** is not used, the description field of the new s-file is left empty.

You can remove the current descriptive text in an s-file by using the **-t** option without a filename. For example, the command:

```
admin -t s.demo.c
```

removes the descriptive text from the s-file *s.demo.c*.

3.7 Printing from an S-file

This section explains how to use the **prs** command to display information contained in an s-file. The **prs** command has a variety of options which control the display format and content.

3.7.1 Using a Data Specification

You can explicitly define the information to be printed from an s-file by using the **-d** option of the **prs** command. The command copies user-specified information to the standard output. The command has the form:

prs -d*spec s.filename*

where -d*spec* is the data specification, and *s.filename* is the name of the s-file from which the information is to be taken.

The data specification is a string of data keywords and text. A data keyword is an uppercase letter, enclosed in colons (:). It represents a value contained in the given s-file. For example, the keyword **:I:** represents the SID of a given version, **:F:** represents the filename of the given s-file, and **:C:** represents the comment line associated with a given version. Data keywords are replaced by these values when the information is printed.

For example, the command:

```
prs -d"version: :I: filename: :F: " s.demo.c
```

may produce the line:

```
version: 2.1 filename: s.demo.c
```

A complete list of the data keywords is given in the section **prs**(CP) in the XENIX *Reference Manual.*

3.7.2 Printing a Specific Version

You can print information about a specific version in a given s-file by using the **-r** option of the **prs** command. The command has the form:

```
prs -rSID s.filename
```

where -r*SID* gives the SID of the desired version, and *s.filename* is the name of the s-file containing the version. For example, the command:

```
prs -r2.1 s.demo.c
```

prints information about version 2.1 in the s-file *s.demo.c*.

If the **-r** option is not specified, the command prints information about the most recently created delta.

3.7.3 Printing Later and Earlier Versions

You can print information about a group of versions by using the **-l** and **-e** options of the **prs** command. The **-l** option causes the command to print information about all versions immediately succeeding the given version. The **-e** option causes the command to print information about all versions immediately preceding the given version. For example, the command:

```
prs -r1.4 -e s.demo.c
```

prints all information about versions which precede version 1.4 (e.g., 1.3, 1.2, and 1.1). The command:

```
prs -r1.4 -l s.abc
```

prints information about versions which succeed version 1.4 (e.g., 1.5, 1.6, and 2.1).

If both options are given, information about all versions is printed.

3.8 Editing by Several Users

The SCCS system allows any number of users to access and edit versions of a given s-file. Since users are likely to access different versions of the s-file at the same time, the system is designed to allow concurrent editing of different versions. Normally, the system allows only one user at a time to edit a given version, but you can allow concurrent editing of the same version by setting the **j** flag in the given s-file.

The following sections explain how to perform concurrent editing, and how to save edited versions when you have retrieved more than one version for editing.

3.8.1 Editing Different Versions

The SCCS system allows several different versions of a file to be edited at the same time. This means a user can edit version 2.1 while another user can edit version 1.1. There is no limit to the number of versions which may be edited at any given time.

When several users edit different versions concurrently, each user must begin work in his own directory. If several users attempt to share a directory, and work on versions from the same s-file, at the same time, the **get** command will refuse to retrieve a version.

3.8.2 Editing a Single Version

You can let a single version of a file be edited by more than one user by setting the **j** flag in the given s-file. The flag causes the **get** command to check the p-file, and create a new proposed SID, if the given version is already being edited.

You can set the flag by using the **-f** option of the **admin** command. For example, the command:

```
admin -fj s.demo.c
```

sets the flag for the s-file, *s.demo.c*.

When the flag is set, the **get** command uses the next available branch SID for each new proposed SID. For example, suppose a user retrieves version 1.4 for editing, in the file *s.demo.c*, and that the proposed version is 1.5. If another user retrieves version 1.4 for editing before the first user has saved his changes, the proposed version for the new user will be 1.4.1.1, since version 1.5 is already proposed and likely to be taken. In no case will a version edited by two separate users, result in a single new version.

3.8.3 Saving a Specific Version

When editing two or more versions of a file, you can direct the **delta** command to save a specific version by using the **-r** option to give the SID of that version. The command has the form:

delta -r*SID s.filename*

where -r*SID* is the SID of the version being saved, and *s.filename* is the name of the s-file to receive the new version. The *SID* may be the SID of the version you have just edited, or the proposed SID for the new version. For example, if you have retrieved version 1.4 for editing (and no version 1.5 exists), both commands:

```
delta -r1.5 s.demo.c
```

and

```
delta -r1.4 s.demo.c
```

save version 1.5.

3.9 Protecting S-files

The SCCS system uses the normal XENIX system file permissions to protect
s-files from changes by unauthorized users. In addition to the XENIX system protections, the SCCS system provides two ways to protect the s-files; the "user list" and the "protection flags". The user list is a list of login names and group IDs of users who are allowed to access the s-file and create new versions of the file. The protection flags are three special s-file flags that define which versions are currently accessible to authorized users. The following sections explain how to set and use the user list and protection flags.

3.9.1 Adding a User to the User List

You can add a user or a group of users to the user list of a given s-file by using the **-a** option of the **admin** command. The **-a** option causes the given name to be added to the user list. The user list defines who may access and edit the versions in the s-file. The command has the form:

```
admin -aname s.filename
```

where -a*name* gives the login name of the user or the group name of a group of users to be added to the list, and *s.filename* gives the name of the s-file to receive the new users. For example, the command:

```
admin -ajohnd -asuex -amarketing s.demo.c
```

adds the users "johnd" and "suex", and the group "marketing" to the user list of the s-file *s.demo.c*.

If you create an s-file without giving the **-a** option, the user list is left empty, and all users may access and edit the files. When you explicitly give a user name or names, only those users can access the files.

3.9.2 Removing a User from a User List

You can remove a user or a group of users from the user list of a given s-file by using the **-e** option of the **admin** command. The option is similar to the **-a** option but performs the opposite operation. The command has the form:

```
admin -ename s.filename
```

where -*ename* is the login name of a user or the group name of a group of users to be removed from the list, and *s.filename* is the name of the s-file from which the names are to be removed. For example, the command:

```
admin -ejohnd -emarketing s.demo.c
```

removes the user "johnd" and the group "marketing" from the user list of the s-file *s.demo.c*.

3.9.3 Setting the Floor Flag

The floor flag, **f**, defines the release number of the lowest version a user may edit in a given s-file. You can set the flag by using the **-f** option of the **admin** command. For example, the command:

```
admin -ff2 s.demo.c
```

sets the floor to release number 2. If you attempt to retrieve any versions with a release number less than 2, an error results.

3.9.4 Setting the Ceiling Flag

The ceiling flag, **c**, defines the release number of the highest version a user may edit in a given s-file. You can set the flag by using the **-f** option of the **admin** command. For example, the command:

```
admin -fc5 s.demo.c
```

sets the ceiling to release number 5. If you attempt to retrieve any versions with a release number greater than 5, an error results.

3.9.5 Locking a Version

The lock flag, **l**, lists by release number all versions in a given s-file which are locked against further editing. You can set the flag by using the **-f** flag of the **admin** command. The flag must be followed by one or more release numbers. Multiple release numbers must be separated by commas (,).

For example, the command:

```
admin -fl3 s.demo.c
```

locks all versions with release number 3 against further editing. The command:

```
admin -fl4,5,9 s.def.h
```

locks all versions with release numbers 4, 5, and 9.

Note that the special symbol "a" may be used to specify all release numbers. The command:

```
admin -fla s.demo.c
```

locks all versions in the file *s.demo.c*.

3.10 Repairing SCCS Files

The SCCS system carefully maintains all SCCS files, making damage to the files very rare. However, damage can result from hardware malfunctions, which cause incorrect information to be copied to the file. The following sections explain how to check for damage to SCCS files, and how to repair the damage or regenerate the file.

3.10.1 Checking an S-file

You can check a file for damage by using the **-h** option of the **admin** command. This option causes the checksum of the given s-file to be computed and compared with the existing sum. An s-file's checksum is an internal value, computed from the sum of all bytes in the file. If the new and existing checksums are not equal, the command displays the following message:

```
corrupted file (co6)
```

indicating damage to the file. For example, the command:

```
admin -h s.demo.c
```

checks the s-file, *s.demo.c*, for damage by generating a new checksum for the file, and comparing the new sum with the existing sum.

You may give more than one filename. If you do, the command checks each file in turn. You may also give the name of a directory, in which case, the command checks all files in the directory.

Since failure to repair a damaged s-file can destroy the file's contents, or make the file inaccessible, it is a good idea to regularly check all s-files for damage.

3.10.2 Editing an S-file

When an s-file is discovered to be damaged, it is a good idea to restore a backup copy of the file from a backup disk, rather than attempting to repair the file. (Restoring a backup copy of a file is described in the XENIX *Operations Guide.*) If this is not possible, the file may be edited using a XENIX text editor.

To repair a damaged s-file, use the description of an s-file given in the section in the XENIX *Reference Manual,* to locate the part of the file which is damaged. Use extreme care when making changes; small errors can cause unwanted results.

3.10.3 Changing an S-file's Checksum

After repairing a damaged s-file, you must change the file's checksum by using the **-z** option of the **admin** command. For example, to restore the checksum of the repaired file *s.demo.c*, enter:

```
admin -z s.demo.c
```

The command computes and saves the new checksum, replacing the old sum.

3.10.4 Regenerating a G-file for Editing

You can create a g-file for editing, without affecting the current contents of the p-file by using the **-k** option of the **get** command. The option has the same affect as the **-e** option, except that the current contents of the p-file remain unchanged. The option is typically used to regenerate a g-file that has been accidentally removed or destroyed, before it has been saved using the **delta** command.

3.10.5 Restoring a Damaged P-file

The **-g** option of the **get** command may be used to generate a new copy of a p-file that has been accidentally removed. For example, the command:

```
get -e -g s.demo.c
```

creates a new p-file entry for the most recent version in *s.demo.c.* If the file *demo.c* already exists, it will not be changed by this command.

3.11 Using Other Command Options

Many of the SCCS commands provide options that control their operation in useful ways. This section describes these options and explains how you may use them to perform useful work.

3.11.1 Getting Help With SCCS Commands

You can display helpful information about an SCCS command by giving the name of the command as an argument to the **help** command. The **help** command displays a short explanation of the command and command syntax. For example, the command:

```
help rmdel
```

displays the message

```
rmdel:
        rmdel -rSID filename...
```

3.11.2 Creating a File With the Standard Input

You can direct **admin** to use the standard input as the source for a new s-file by using the **-i** option without a filename. For example, the command:

```
admin -i s.demo.c <demo.c
```

causes **admin** to create a new s-file named *s.demo.c,* which uses the text file *demo.c* as its first version.

This method of creating a new s-file is typically used to connect **admin** to a pipe. For example, the command:

```
cat mod1.c mod2.c | admin -i s.mod.c
```

creates a new s-file, *s.mod.c,* which contains the first version of the concatenated files *mod1.c* and *mod2.c.*

3.11.3 Starting At a Specific Release

The **admin** command normally starts numbering versions with release number 1. You can direct the command to start with any given release number by using the **-r** option. The command has the form:

admin -r*rel-num s.filename*

where *-rrel-num* gives the value of the starting release number, and *s.filename* is the name of the s-file to be created. For example, the command:

```
        admin -idemo.c -r3 s.demo.c
```

starts with release number 3. The first version is 3.1.

3.11.4 Adding a Comment to the First Version

You can add a comment to the first version of file by using the **-y** option of the **admin** command when creating the s-file. For example, the command:

```
        admin -idemo.c -y"George Wheeler" s.demo.c
```

inserts the comment "George Wheeler" in the new s-file, *s.demo.c*.

The comment may be any combination of letters, digits, and punctuation symbols. If spaces are used, the comment must be enclosed in double quotes. The complete command must fit on one line.

If the **-y** option is not used when creating an s-file, a comment of the following form:

```
        date and time created YY/MM/DD HH:MM:SS by logname
```

is automatically inserted.

3.11.5 Suppressing Normal Output

You can suppress the normal display of messages created by the **get** command by using the **-s** option. The option prevents information, such as the SID of the retrieved file, from being copied to the standard output. The option does not suppress error messages.

The **-s** option is often used with the **-p** option to pipe the output of the **get** command to other commands. For example, the command:

```
        get -p -s s.demo.c | lpr
```

copies the most recent version in the s-file, *s.demo.c*, to the line printer.

You can also suppress the normal output of the **delta** command by using the **-s** option. This option suppresses all output normally directed to the standard output, except for the normal comment prompt.

3.11.6 Including and Excluding Deltas

You can explicitly define which deltas you wish to include, and which you wish to exclude, when creating a g-file, by using the **-i** and **-x** options of the **get** command.

The **-i** option causes the command to apply the given deltas when constructing a version. The **-x** option causes the command to ignore the given deltas when constructing a version. Both options must be followed by one or more SIDs. If multiple SIDs are given, they must be separated by commas (,). A range of SIDs may be given, by separating two SIDs with a hyphen (-). For example, the command:

```
get -i1.2,1.3 s.demo.c
```

causes deltas 1.2 and 1.3 to construct the g-file. The command:

```
get -x1.2-1.4 s.demo.c
```

causes deltas 1.2 through 1.4 to be ignored when constructing the file.

The **-i** option is useful if you wish to automatically apply changes to a version while retrieving it for editing. For example, the command:

```
get -e -i4.1 -r3.3 s.demo.c
```

retrieves version 3.3 for editing. When the file is retrieved, the changes in delta 4.1 are automatically applied to it, making the g-file the same as if version 3.3 had been edited by hand, using the changes in delta 4.1. These changes can be saved immediately by issuing a **delta** command. No editing is required.

The **-x** option is useful if you wish to remove changes performed on a given version. For example, the command:

```
get -e -x1.5 -r1.6 s.demo.c
```

retrieves version 1.6 for editing. When the file is retrieved, the changes in delta 1.5 are automatically left out of it, making the g-file the same as if version 1.4 had been changed according to delta 1.6 (with no intervening delta 1.5). These changes can be saved immediately by issuing a **delta** command. No editing is required.

When deltas are included or excluded using the **-i** and **-x** options, **get** compares them with the deltas that are normally used in constructing the given version. If two deltas attempt to change the same line of the retrieved file, the command displays a warning message. The message shows the range of lines in which the problem may exist. Corrective action, if required, is the responsibility of the user.

3.11.7 Listing the Deltas of a Version

You can create a table showing the deltas required to create a given version by using the **-l** option. This option causes the **get** command to create an l-file which contains the SIDs of all deltas used to create the given version.

The option is typically used to create a history of a given version's development. For example, the command:

```
get -l s.demo.c
```

creates a file named *l.demo.c*, containing the deltas required to create the most recent version of *demo.c*.

You can display the list of deltas required to create a version by using the **-lp** option. The **-lp** option performs the same function as the **-l** option, except it copies the list to the standard output file. For example, the command:

```
get -lp -r2.3 s.demo.c
```

copies the list of deltas required to create version 2.3 of *demo.c* to the standard output.

Note that the **-l** option may be combined with the **-g** option to create a list of deltas without retrieving the actual version.

3.11.8 Mapping Lines to Deltas

You can map each line in a given version to its corresponding delta by using the **-m** option of the **get** command. This option causes each line in a g-file to be preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the beginning of the line by a tab character. The **-m** option is typically used to review the history of each line in a given version.

3.11.9 Naming Lines

You can name each line in a given version with the current module name (i.e., the value of the **%M%** keyword) by using the **-n** option of the **get** command. This option causes each line of the retrieved file to be preceded by the value of the **%M%** keyword and a tab character.

The **-n** option is typically used to indicate that a given line is from the given file. When both the **-m** and **-n** options are specified, each line begins with the **%M%** keyword.

3.11.10 Displaying a List of Differences

You can display a detailed list of the differences between a new version of a file and the previous version by using the **-p** option of the **delta** command. This option causes the command to display the differences, in a format similar to the output of the XENIX **diff** command.

3.11.11 Displaying File Information

You can display information about a given version by using the **-g** option of the **get** command. This option suppresses the actual retrieval of a version and causes only the information about the version, such as the SID and size, to be displayed.

The **-g** option is often used with the **-r** option to check for the existence of a given version. For example, the command:

```
get -g -r4.3 s.demo.c
```

displays information about version 4.3 in the s-file, *s.demo.c*. If the version does not exist, the command displays an error message.

3.11.12 Removing a Delta

You can remove a delta from an s-file by using the **rmdel** command. The command has the form:

rmdel -r*SID s.filename*

where -r*SID* gives the SID of the delta to be removed, and *s.filename* is the name of the s-file from which the delta is to be removed. The delta must be the most recently created delta in the s-file. Furthermore, the user must have write permission in the directory containing the s-file, and must either own the s-file or be the user who created the delta.

For example, the command:

```
rmdel -r2.3 s.demo.c
```

removes delta 2.3 from the s-file, *s.demo.c*.

The **rmdel** command will refuse to remove a protected delta, that is, a delta whose release number is below the current floor value, above the current ceiling value, or equal to a current locked value (see the section "Protecting S-files" given earlier in this chapter). The command will also refuse to remove a delta which is currently being edited.

The **rmdel** command should be reserved for those cases in which incorrect, global changes were made to an s-file.

Note that **rmdel** changes the type indicator of the given delta from "D" to "R". A type indicator defines the type of delta. Type indicators are described in detail in the section in the XENIX *Reference Manual*.

3.11.13 Searching for Strings

You can search for strings in files created from an s-file by using the **what** command. This command searches for the symbol #(@) (the current value of the **%Z%** keyword), in the given file. It then prints, on the standard output, all text immediately following the symbol, up to the next double quote ("), greater than (>), backslash (\), newline, or (non-printing) NULL character. For example, if the s-file, *s.demo.c*, contains the following line:

```
char id[]="%Z% %M%:%I%";
```

and the command:

```
get -r3.4 s.prog.c
```

is executed, then the command:

```
what prog.c
```

displays:

```
prog.c:
        prog.c:3.4
```

You may also use **what** to search files that have not been created by SCCS commands.

3.11.14 Comparing SCCS Files

You can compare two versions from a given s-file by using the **sccsdiff** command. This command prints the differences between two versions of the s-file on the standard output. The command has the form:

sccsdiff -r*SID1* -r*SID2* *s.filename*

where -r*SID1* and -r*SID2* give the SIDs of the versions to be compared, and *s.filename* is the name of the s-file containing the versions. The version SIDs must be given in the order in which they were created. For example, the command:

```
sccsdiff -r3.4 -r5.6 s.demo.c
```

displays the differences between versions 3.4 and 5.6. The differences are displayed in a form similar to the XENIX **diff** command.

Chapter 4

lint: A C Program Checker

4.1 Introduction 4-1

4.2 Invoking lint 4-1

4.3 Options 4-2

4.4 Checking for Unused Variables and Functions 4-3

4.5 Checking Local Variables 4-4

4.6 Checking for Unreachable Statements 4-5

4.7 Checking for Infinite Loops 4-6

4.8 Checking Function Return Values 4-6

4.9 Checking for Unused Return Values 4-7

4.10 Checking Types 4-7

4.11 Checking Type Casts 4-8

4.12 Checking for Nonportable Character Use 4-9

4.13 Checking for Assignment of longs to ints 4-9

4.14 Checking for Strange Constructions 4-9

4.15 Checking for Use of Older C Syntax 4-11

4.16 Checking Pointer Alignment 4-12

4.17 Checking Expression Evaluation Order 4-12

4.18 Embedding Directives 4-13

4.19 Checking For Library Compatibility 4-14

4.1 Introduction

This chapter explains how to use the C program checker **lint**(CP). The program examines C source files and warns of errors or misconstructions that may cause errors during compilation of the file or during execution of the compiled file.

In particular, **lint** checks for:

- Unused functions and variables
- Unknown values in local variables
- Unreachable statements and infinite loops
- Unused and misused return values
- Inconsistent types and type casts
- Mismatched types in assignments
- Nonportable and old-fashioned syntax
- Strange constructions
- Inconsistent pointer alignment and expression evaluation order

The **lint** program and the C compiler are generally used together to check and compile C language programs. Although the C compiler rapidly and efficiently compiles C language source files, it does not perform the sophisticated type and error checking required by many programs. The **lint** program, on the other hand, provides thorough checking of source files without compiling.

4.2 Invoking lint

You can invoke **lint** by typing its name at the shell command line. The command has the form:

lint [*option...*] *filename ... lib ...*

where *option* is a command option that defines how the checker should operate, *filename* is the name of the C language source file to be checked, and *lib* is the name of a library to check. You can give more than one option, filename, or library name in the command as long as you use spaces to separate them. If you give two or more filenames, **lint** assumes that the files form a complete program and checks the files accordingly. For example, the command:

```
lint main.c add.c
```

treats *main.c* and *add.c* as two parts of a complete program.

If **lint** discovers errors or inconsistencies in a source file, it produces messages describing the problem. The message has the form:

filename (*num*): *description*

where *filename* is the name of the source file containing the problem, *num* is the number of the line in the source containing the problem, and *description* is a description of the problem. For example, the message:

```
main.c (3): warning: x unused in function main
```

shows that the variable *x*, defined in line three of the source file *main.c*, is not used anywhere in the file.

4.3 Options

The options available to you may be classed into two categories: those that instruct **lint** to suppress certain kinds of complaints, and those that alter the behavior of **lint**. The following list summarizes both kinds of options:

Suppressive Options

– a	Suppresses complaints about assignments of long values to variables that are not long.
– b	Suppresses complaints about **break** statements that cannot be reached (programs produced by **lex** or **yacc** will often result in a large number of such complaints).
– c	Suppresses complaints about casts that have questionable portability.
– h	Does not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

– u Suppresses complaints about functions and external variables used and not defined, or defined and not used (this option is suitable for running **lint** on a subset of files of a larger program).

– v Suppresses complaints about unused arguments in functions.

– x Does not report variables referred to by external declarations but never used.

Other Options

– n Does not check compatibility against either the standard or the portable lint library.

– p Attempts to check portability to other dialects of C.

– *llibrary* Checks function definitions in the specified lint *library*. For example, **–lm** causes the library *llibm.ln* to be searched.

4.4 Checking for Unused Variables and Functions

The **lint** program checks for unused variables and functions by seeing if each declared variable and function is used at least once in the source file. The program considers a variable or function used if the name appears in at least one statement. It is not considered used if it only appears on the left side of an assignment. For example, in the following program fragment:

```
main ()
{
        int x,y,z;

        x=1; y=2; z=x+y;
```

the variables *x* and *y* are considered used, but variable *z* is not.

Unused variables and functions often occur during the development of large programs. It is not uncommon for a programmer to remove all references to a variable or function from a source file, but forget to remove its declaration. Such unused variables and functions rarely cause working programs to fail, but do make programs harder to understand and change. Checking for unused variables and functions can also help you find variables or functions that you intended to use but accidentally have left out of the program.

Note that the **lint** program does not report a variable or function unused if it is explicitly declared with the **extern** storage class. Such a variable or function is assumed to be used in another source file.

You can direct **lint** to ignore all the external declarations in a source file by using the **−x** (for "external") option. This option causes the program checker to skip any line that begins with the **extern** storage class. The **−x** option is typically used to save time when checking a program, especially if all external declarations are known to be valid.

Some programming styles require functions that perform closely related tasks to have the same number and type of arguments, regardless of whether these arguments are used. Under normal operation, **lint** reports any argument not used as an unused variable. You can direct **lint** to ignore unused arguments by using the **−v** option.

The **−v** option causes **lint** to ignore all unused function arguments except for those declared with **register** storage class. The program considers unused arguments of this class to be a preventable waste of the register resources of the computer.

You can direct **lint** to ignore all unused variables and functions by using the **−u** (for "unused") option. This option prevents **lint** from reporting variables and functions it considers unused.

The **−u** option is typically used when checking a source file that contains just a portion of a large program. Such source files usually contain declarations of variables and functions that are intended to be used in other source files and are not explicitly used within the file. Since **lint** can only check the given file, it assumes that such variables or functions are unused and and reports them as errors whenever the **−u** option is not given.

4.5 Checking Local Variables

The **lint** program checks all local variables to ensure that they are set to a value before being used. Since local variables have either automatic or register storage class, their values at the start of the program or function cannot be known. Using such a variable before assigning a value to it is an error.

The **lint** program checks the local variables by searching for the first assignment in which the variable receives a value, and for the first statement or expression in which the variable is used. If the first assignment appears later than the first use, **lint** considers the variable inappropriately used. For example, in the program fragment

```
char c;

if ( c != EOT )
        c = getchar();
```

lint warns that the the variable *c* is used before it is assigned.

If a variable is used in the same statement in which it is assigned for the first time, **lint** determines the order of evaluation of the statement and displays an appropriate message. For example, in the program fragment

```
int i,total;

scanf("%d", &i);
total = total + i;
```

lint warns that the variable *total* is used before it is set, since it appears on the right side of the same statement that assigns its first value.

Static and external variables are always initialized to zero before program execution begins, so **lint** does not report such variables if they are used before being set to a value.

4.6 Checking for Unreachable Statements

The **lint** program checks for unreachable statements. Unreachable statements are unlabeled statements that immediately follow a **goto**, **break**, **continue**, or **return** statement. During execution of a program, the unreachable statements never receive execution control and therefore are considered wasteful. For example, in the program fragment:

```
int x,y;

return (x+y);
exit (1);
```

the function call **exit** after the return statement is unreachable.

Unreachable statements are common when developing programs containing large case constructions, or loops containing break and continue statements. Such statements are wasteful and should be removed when convenient.

During normal operation, **lint** reports all unreachable break statements. Unreachable break statements are relatively common (some programs created by the **yacc** and **lex** programs contain hundreds), so it may be desirable to suppress these reports. You can direct **lint** to suppress the reports by using the **−b** option.

Note that **lint** assumes that all functions eventually return control, so it does not report as unreachable any statement that follows a function that takes control and never returns it. For example, in the program fragment

```
exit (1);
return;
```

the call to **exit** causes the **return** statement to become an unreachable statement, but **lint** does not report it as such.

4.7 Checking for Infinite Loops

The **lint** program checks for infinite loops and for loops that are never executed. For example, the statements:

```
while (1) { }
```

and:

```
for (;;) {}
```

are both considered infinite loops. The statements:

```
while (0) { }
```

and:

```
for (0;0;) { }
```

will be reported as never executed.

Although some valid programs have such loops, they are generally considered errors.

4.8 Checking Function Return Values

The **lint** program checks to ensure that a function returns a meaningful value if a return value is expected. Some functions return values that are never used. Some programs incorrectly use function values that have never been returned. **lint** addresses these problems in a number of ways.

Within a function definition, the appearance of both:

```
return (expr);
```

and:

```
return ;
```

statements is cause for alarm. In this case, **lint** produces the following error message:

warning: function *filename* has return(e); and return;

It is difficult to detect when a function return is implied by the flow of control reaching the end of the given function. This is demonstrated with a simple example:

```
f (a)
{
        if (a)
                return (3);
        g();
}
```

If *a* is false, then *f*() will call the function *g*() and then return with no defined return value. This will trigger a report from **lint**. If *g*(), like *exit*(), never returns, the message will still be produced when in fact nothing is wrong. In practice, potentially serious bugs can be discovered with this feature. It also accounts for a substantial fraction of the undeserved error messages produced by **lint**.

4.9 Checking for Unused Return Values

The **lint** program checks for cases where a function returns a value, but the value is rarely if ever used. **lint** considers functions that return unused values to be inefficient, and functions that return rarely used values to be a result of bad programming style.

lint also checks for cases where a function does not return a value but the value is used anyway. This is considered a serious error.

4.10 Checking Types

lint enforces the type checking rules of C more strictly than the C compiler. The additional checking occurs in four major areas:

1. Across certain binary operators and implied assignments
2. At the structure selection operators
3. Between the definition and uses of functions
4. In the use of enumerations

There are a number of operators that have an implied balancing between types of operands. The assignment, conditional, and relational operators have this property. The argument of a **return** statement, and expressions used in initialization also suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of x's can be intermixed with pointers to x's.

The type checking rules also require that, in structure references, the left operand of a pointer arrow symbol (->) must be a pointer to a structure, the left operand of a period (.) must be a structure, and the right operand of these operators must be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Pointers can also be matched with the associated arrays. Aside from these relaxations in type checking, all actual arguments must agree in type with their declared counterparts.

lint checks to ensure that enumeration variables or members are not mixed with other types or other enumerations. It also ensures that the only operations applied to enumerated variables are assignment (=), initialization, equals (==), and not-equals (!=). Enumerations may also be function arguments and return values.

4.11 Checking Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment:

```
p = 1 ;
```

where *p* is a character pointer. **lint** reports this as suspect. However, in the assignment:

```
p = (char *)1 ;
```

a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his intentions. On the other hand, if this code is moved to another machine, it should be looked at carefully. The **-c** option controls the printing of comments about casts. When **-c** is in effect, casts are not checked, and all legal casts are passed without comment, no matter how strange the type mixing seems to be.

4.12 Checking for Nonportable Character Use

lint flags certain comparisons and assignments as illegal or nonportable. For example, the fragment:

```
char c;
 .
 .
 .
if( (c = getchar()) < 0 ) ...
```

works on some machines, but fails on machines where characters always take on positive values. In this case, **lint** issues the message:

```
nonportable character comparison
```

The solution is to declare *c* an integer, since **getchar** is actually returning integer values.

A similar issue arises with bitfields. When assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true where on some machines bitfields are considered as signed quantities. Although a 2-bit field with **int** type cannot hold the value 3, a 2-bit field with **unsigned** type can.

4.13 Checking for Assignment of longs to ints

Problems may arise from the assignment of **long** values to **int** values, because of a loss in accuracy in the assignment. This may happen in programs that have been incompletely converted by changing type definitions with **typedef**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to integer values, losing accuracy. Since there are a number of legitimate reasons for assigning longs to integers, you may wish to suppress detection of these assignments by using the **−a** option.

4.14 Checking for Strange Constructions

Several perfectly legal but somewhat strange constructions are flagged by **lint**. The generated messages encourage better code quality, clearer style, and may even point out bugs. For example, in the statement

```
*p++ ;
```

the star (*) does nothing, so **lint** prints:

```
null effect
```

The program fragment:

```
unsigned x ;
if (x < 0) ...
```

is also considered strange since the test will never succeed.

Similarly, the test

```
if (x > 0)
```

is equivalent to

```
if (x != 0)
```

which may not be the intended action. In these cases, **lint** prints the message

```
degenerate unsigned comparison
```

If you use:

```
if( 1 != 0 ) ...
```

then **lint** reports

```
constant in conditional context
```

since the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs that arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements:

```
if( x&077 == 0 ) ...
```

or:

```
x<< 2 + 40
```

probably do not do what is intended. The best solution is to place parentheses around such expressions. **lint** encourages this by printing an appropriate message.

Finally, **lint** checks variables that are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered bad style, usually unnecessary, and frequently points out a bug.

If you do not wish these heuristic checks, you can suppress them by using the **−h** option.

4.15 Checking for Use of Older C Syntax

lint checks for older C constructions. These fall into two classes: assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =−, ...) can cause ambiguous expressions, such as:

```
a =−1 ;
```

which could be taken as either:

```
a =− 1 ;
```

or:

```
a = −1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer, and preferred operators (e.g., +=, −=) have no such ambiguities. To encourage the abandonment of the older forms, **lint** checks for occurrences of these old-fashioned operators.

A similar issue arises with initialization. The older language allowed:

```
int x 1 ;
```

to initialize *x* to 1. This causes syntactic difficulties. For example:

```
int x (−1) ;
```

looks somewhat like the beginning of a function declaration:

```
int x (y) { ...
```

and the compiler must read past *x* to determine what the declaration really is. The problem is even more perplexing when the initializer involves a macro. The current C syntax places an equal sign between the variable and the initializer:

```
int x = −1 ;
```

This form is free of any possible syntactic ambiguity.

4.16 Checking Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due to alignment restrictions. For example, on some machines it is reasonable to assign integer pointers to double pointers, since double precision values may begin on any integer boundary. On other machines, however, double precision values must begin on even word boundaries; thus, not all such assignments make sense. **lint** tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message:

```
possible pointer alignment problem
```

results from this situation.

4.17 Checking Expression Evaluation Order

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine-dependent. For example, on machines in which the stack runs backwards, function arguments will probably best be evaluated from right to left; on machines with a stack running forward, left to right is probably best. Function calls embedded as arguments of other functions may or may not be treated in the same way as ordinary arguments. Similar issues arise with other operators that have side effects, such as the assignment operators and the increment and decrement operators.

To ensure maximum efficiency of C on a particular machine, the C language leaves the order of evaluation of complicated expressions up to the compiler. Various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is undefined.

lint checks for the important special case where a simple scalar variable is affected. For example, the statement:

```
a[i] = b[i++];
```

will draw the comment:

```
warning: i evaluation order undefined
```

4.18 Embedding Directives

There are occasions when the programmer is smarter than **lint**. There may be valid reasons for illegal type casts, functions with a variable number of arguments, and other constructions that **lint** finds objectionable. Moreover, as specified in the above sections, the flow of control information produced by **lint** often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Some way of communicating with **lint**, typically to turn off its output, is desirable. Therefore, a number of words are recognized by **lint** when they are embedded in comments in a C source file. These words are called directives. **lint** directives are invisible to the compiler.

The first directive discussed concerns flow of control information. If a particular place in the program cannot be reached, this can be asserted at the appropriate spot in the program with the directive:

```
/* NOTREACHED */
```

Similarly, if you desire to turn off strict type checking for the next expression, use the directive:

```
/* NOSTRICT */
```

The situation reverts to the previous default after the next expression. The **−v** option can be turned on for one function with the directive:

```
/* ARGSUSED */
```

Comments about a variable number of arguments in calls to a function can be turned off by preceding the function definition with the directive:

```
/* VARARGS */
```

In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. You can define the number of arguments to be checked by placing a digit (giving this number) immediately after the **VARARGS** keyword. For example,

```
/* VARARGS2 */
```

causes only the first two arguments to be checked. Finally, the directive:

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file, which is discussed in the next section.

4.19 Checking For Library Compatibility

lint accepts certain library directives, such as:

```
-ly
```

and tests the source files for compatibility with these libraries. This testing is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive:

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. These definitions indicate whether a function returns a value, what type a function's return type is, and the number and types of arguments expected by the function. The **VARARGS** and **ARGSUSED** directives can be used to specify features of the library functions.

lint library files are processed almost exactly like ordinary source files. The only difference is that functions that are defined in a library file, but are not used in a source file, draw no comments. **lint** does not simulate a full library search algorithm, and checks to see if the source files contain redefinitions of library routines.

By default, **lint** checks the programs it is given against a standard library file, which contains descriptions of the programs that are normally loaded when a C program is run. When the **−p** option is in effect, the portable library file is checked. This library contains descriptions of the standard I/O library routines which are expected to be portable across various machines. The **−n** option can be used to suppress all library checking.

Chapter 5

lex: A Lexical Analyzer

5.1 Introduction 5-1

5.2 lex Regular Expressions 5-3

5.3 Invoking **lex** 5-4

5.4 Specifying Character Classes 5-4

5.5 Specifying an Arbitrary Character 5-5

5.6 Specifying Optional Expressions 5-5

5.7 Specifying Repeated Expressions 5-6

5.8 Specifying Alternation and Grouping 5-6

5.9 Specifying Context Sensitivity 5-6

5.10 Specifying Expression Repetition 5-7

5.11 Specifying Definitions 5-7

5.12 Specifying Actions 5-8

5.13 Handling Ambiguous Source Rules 5-12

5.14 Specifying Left Context Sensitivity 5-15

5.15 Specifying Source Definitions 5-17

5.16 lex and yacc 5-19

5.17 Specifying Character Sets 5-23

5.18 Source Format 5-24

5.1 Introduction

lex(CP) is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem-oriented specification for character string matching, and produces a C program that recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to **lex**. The **lex** code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings, program sections provided by the user are executed. The **lex** source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by **lex**, the corresponding fragment is executed.

The user supplies the additional code needed to complete his tasks, including code written by other generators. The program that recognizes the expressions is generated in the from the user's C program fragments. **lex** is not a complete language, but rather a generator representing a new language feature added on top of the C programming language.

lex turns the user's expressions and actions (called *source* in this chapter) into a C program named *yylex*. The *yylex* program recognizes expressions in a stream (called input in this chapter) and performs the specified actions for each expression as it is detected.

Consider a program to delete from the standard input all blanks or tabs at the ends of lines. The following lines:

```
%%
[\t]+$ ;
```

are all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one other rule. This rule contains a regular expression that matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates one or more of the previous item; and the dollar sign ($) indicates the end of the line. No action is specified, so the program generated by **lex** will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[\t]+$ ;
[\t]+   printf(" ");
```

The finite automaton generated for this source scans for both rules at once, checks for the termination of a string of blanks or tabs; whether or not there is a newline character; and then executes the desired rule's action.

The first rule matches all strings of blanks or tabs at the end of lines, and the second rule matches all remaining strings of blanks or tabs.

lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. **lex** can also be used with a parser generator to perform the lexical analysis phase; it is especially easy to interface **lex** and **yacc**(CP). **lex** programs recognize only regular expressions; **yacc** writes parsers that accept a large class of context-free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of **lex** and **yacc** is often appropriate. When used as a preprocessor for a later parser generator, **lex** is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. Additional programs, written by other generators or by hand, can be added easily to programs written by **lex**. **yacc** users will realize that the name *yylex* is what **yacc** expects its lexical analyzer to be named, so that the use of this name by **lex** simplifies interfacing.

lex generates a deterministic finite automaton from the regular expressions in the source. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a **lex** program to recognize and partition an input stream is proportional to the length of the input. The number or the complexity of **lex** rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by **lex**.

In a program written by **lex**, the user's fragments (representing the actions to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

lex is not limited to source that can be interpreted on the basis of one character lookahead. For example, if there are two rules, one looking for *ab* and another for *abcdefg*, and the input stream is *abcdefh*, **lex** will recognize *ab* and leave the input pointer just before *cd*. Such backup is more costly than the processing of simpler languages.

5.2 lex Regular Expressions

A regular expression specifies a set of strings to be matched. It contains text characters (that match the corresponding characters in the strings being compared) and operator characters (these specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters. Thus, the regular expression:

```
integer
```

matches the string *integer* wherever it appears and the expression:

```
a57D
```

looks for the string *a57D*.

The operator characters are:

```
"\[]^-?.*+|()$/{}%<>
```

If any of these characters are to be used literally, they need to be quoted individually with a backslash (\), or as a group within quotation marks ("). The quotation mark operator (") indicates that whatever is contained between a pair of quotation marks is to be taken as text characters. Thus:

```
xyz"++"
```

matches the string *xyz++* when it appears. Note that part of a string may be quoted. It is harmless, but unnecessary, to quote an ordinary text character; the expression:

```
"xyz++"
```

is the same as the one above. Thus by quoting every nonalphanumeric character being used as a text character, you do not have to remember the above list of current operator characters.

An operator character may also be turned into a text character by preceding it with a backslash (\) as in:

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions. The quoting mechanism can also be used to get a blank into an expression. Normally, blanks or tabs end a rule. Any blank character not contained within brackets must be quoted. Several normal C escapes with the backslash (\) are recognized:

```
\n     newline
```

\t	tab
\b	backspace
\\	backslash

Since newline is illegal in an expression, a *\n* must be used; it is not required to escape tab and backspace. Note that every character is always a text character. Exceptions to this are blanks, tabs, newlines and the operator characters shown in the list above.

5.3 Invoking lex

There are two steps in compiling a **lex** source program. First, the **lex** source must be turned into a generated program in the host general purpose language. Then this program must be compiled and loaded, usually with a library of **lex** subroutines. The generated program is in a file named *lex.yy.c*. The I/O library is defined in terms of the C standard library.

The library is accessed by the loader flag *−ll*. So an appropriate set of commands is

```
lex source
cc lex.yy.c −ll
```

The resulting program is placed in the file *a.out* for later execution. To use **lex** with **yacc** see the section "lex and yacc" in this chapter and Chapter 6, "yacc: A Compiler-Compiler"". Although the default **lex** I/O routines use the C standard library, the **lex** automata themselves do not. If private versions of *input*(), *output*(), and *unput*() are given, the library can be avoided.

5.4 Specifying Character Classes

Classes of characters can be specified using brackets: [and]. The construction:

```
[abc]
```

matches a single character, which may be *a*, *b*, or *c*. Within square brackets, most operator meanings are ignored. Only three characters are special: these are the backslash (\), the dash (-), and the caret (^). The dash character indicates ranges. For example:

```
[a-z0-9<>_]
```

indicates the character class containing all the lowercase letters, the digits, the angle brackets, and underline. Ranges may be given in either order.

Using the dash between any pair of characters that are not both uppercase letters, both lowercase letters, or both digits is implementation dependent and causes a warning message. If it is desired to include the dash in a character class, it should be first or last; thus:

```
[-+0-9]
```

matches all the digits and the plus and minus signs.

In character classes, the caret (^) operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus:

```
[^abc]
```

matches all characters except *a*, *b*, or *c*, including all special or control characters; or:

```
[^a-zA-Z]
```

is any character which is not a letter. The backslash (\ ) provides an escape mechanism within character class brackets, so that characters can be entered literally by preceding them with this character.

5.5 Specifying an Arbitrary Character

To match almost any character, the period (.) designates the class of all characters except a newline. Escaping into octal is possible although non-portable. For example:

```
[\40-\176]
```

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

5.6 Specifying Optional Expressions

The question mark (?) operator indicates an optional element of an expression. Thus:

```
ab?c
```

matches either *ac* or *abc*. Note that the meaning of the question mark here differs from its meaning in the shell.

5.7 Specifying Repeated Expressions

Repetitions of classes are indicated by the asterisk (*) and plus (+) operators. For example:

```
a*
```

matches any number of consecutive *a* characters, including zero; while *a+* matches one or more instances of *a*. For example:

```
[a-z]+
```

matches all strings of lowercase letters, and:

```
[A-Za-z][A-Za-z0-9]*
```

matches all alphanumeric strings with a leading alphabetic character. Note that this is a typical expression for recognizing identifiers in computer languages.

5.8 Specifying Alternation and Grouping

The vertical bar (|) operator indicates alternation. For example:

```
(ab|cd)
```

matches either *ab* or *cd*. Note that parentheses are used for grouping, although they are not necessary at the outside level. For example:

```
ab|cd
```

would have sufficed in the preceding example. Parentheses should be used for more complex expressions, such as:

```
(ab|cd+)?(ef)*
```

which matches such strings as *abefef*, *efefef*, *cdef*, and *cddd*, but not *abc*, *abcd*, or *abcdef*.

5.9 Specifying Context Sensitivity

lex recognizes a small amount of surrounding context. The two simplest operators for this are the caret (ˆ) and the dollar sign ($). If the first character of an expression is a caret, then the expression is only matched at the beginning of a line (after a newline character, or at the beginning of the input stream). This can never conflict with the other meaning of the caret, complementation of character classes, since complementation only applies within brackets. If the very last character is a dollar sign, the expression is only matched at the end of a line (when immediately followed by newline). The latter operator is a special case of the slash (/) operator, which indicates trailing context. The expression:

```
ab/cd
```

matches the string *ab*, but only if followed by *cd*. Thus:

```
ab$
```

is the same as:

```
ab/\n
```

Left context is handled in **lex** by specifying start conditions as explained in section 5.14, "Specifying Left Context Sensitivity". If a rule is only to be executed when the **lex** automaton interpreter is in start condition *x*, the rule should be enclosed in angle brackets:

```
<x>
```

If we considered start condition ONE as being at the beginning of a line, then the caret (^) operator would be equivalent to:

```
<ONE>
```

Start conditions are explained in detail later in this chapter.

5.10 Specifying Expression Repetition

The curly braces ({ and }) specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name). For example:

```
{digit}
```

looks for a predefined string named *digit* and inserts it at that point in the expression.

5.11 Specifying Definitions

The definitions are given in the first part of the **lex** input, before the rules. In contrast:

```
a{1,5}
```

looks for 1 to 5 occurrences of the character *a*.

Finally, an initial percent sign (%) is special, since it is the separator for **lex** source segments.

5.12 Specifying Actions

When an expression is matched by a pattern of text in the input, **lex** executes the corresponding action. This section describes some features of **lex** which aid in writing actions. Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the **lex** user who wishes to absorb the entire input, without producing any output, must provide rules to match everything. When **lex** is being used with **yacc**, this is considered to be the normal situation. You may consider that actions are done instead of copying the input to the output; thus, a rule which merely copies can be omitted.

One of the simplest things that can be done is to ignore the input. Specifying a C null statement *;* as an action causes this result. A frequent rule is:

```
[\t\n] ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored.

Another easy way to avoid writing actions is to use the repeat action character, |, which indicates that the action for this rule is the action for the next rule. The previous example could also have been written:

```
" "             |
"\t"            |
"\n"            ;
```

with the same result, although in a different style. The quotes around *\n* and *\t* are not required.

In more complex actions, you often want to know the actual text that matched some expression like:

```
[a-z]+
```

lex leaves this text in an external character array named *yytext*. Thus, to print the name found, a rule like:

```
[a-z]+ printf("%s", yytext);
```

prints the string in *yytext*. The C function **printf**(S) accepts a format argument and data to be printed; in this case, the format is *print string* where the percent sign (%) indicates data conversion, and the *s* indicates string type, and the data are the characters in *yytext*. This places the matched string on the output. This action is so common that it may be written as ECHO. For example:

```
[a-z]+ ECHO;
```

is the same as the preceding example. Since the default action is just to print the characters found, one might ask why give a rule which merely specifies the default action? Such rules are often required to avoid matching some other rule that is not desired. For example, if there is a rule that matches *read*, it will normally match the instances of *read* contained in *bread* or *readjust*; to avoid this, a rule of the form:

```
[a–z]+
```

is needed. This is explained further below.

Sometimes it is more convenient to know the end of what has been found; hence **lex** also provides a count of the number of characters matched in the variable, *yyleng*. To count both the number of words and the number of characters in words in the input, you might enter:

```
[a–zA–Z]+    {words++; chars += yyleng;}
```

which accumulates the number of characters in the words recognized, and places the result in the variable *chars*. The last character in the matched string can be accessed with:

```
yytext[yyleng–1]
```

Occasionally, a **lex** action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation. First, *yymore*() can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string will overwrite the current entry in *yytext*. Second, *yyless*(n) may be called to indicate that not all the characters matched by the currently successful expression are needed right now. The argument *n* indicates the number of characters in *yytext* to be retained. Further characters previously matched are returned to the input. This provides the same sort of lookahead offered by the slash (/) operator, but in a different form.

For example, consider a language that defines a string as a set of characters between quotation marks ("), and provides that in order to include a quotation mark in a string, it must be preceded by a backslash (\). The regular expression that matches this is somewhat confusing, so it might be preferable to enter:

```
\"[^"]* {
        if (yytext[yyleng-1] == '\\')
          yymore();
        else
          ... normal user processing
        }
```

which, when faced with a string such as:

```
"abc\"def"
```

will first match the five characters:

```
"abc\
```

and then the call to *yymore*() will cause the next part of the string:

```
"def
```

to be tacked on the end. Note that the final quotation mark terminating the string should be picked up in the code labeled normal processing.

The function *yyless*() might be used to reprocess text in various circumstances. Consider the problem in the older C syntax of distinguishing the ambiguity of =−*a*. Suppose it is desired to treat this as =− *a* and to then print a message. The following rule might apply:

```
=-[a-zA-Z] {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-1);
        ... action for =- ...
        }
```

which prints a message, returns the letter after the operator to the input stream, and treats the operator as =−.

Alternatively, it might be desired to treat this as = *−a*. To do this, just return the minus sign as well as the letter to the input. The following performs the interpretation:

```
=-[a-zA-Z]  {
        printf("Operator (=-) ambiguous\n");
        yyless(yyleng-2);
        ... action for = ...
        }
```

Note that the expressions for the two cases could also be written as:

```
=-/[A-Za-z]
```

in the first case and:

```
=/-[A-Za-z]
```

in the second. No backup would be required in the rule action shown above. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of =−*3*, however, makes:

```
=-/[^ \t\n]
```

a better rule.

In addition to these routines, **lex** also permits access to the I/O routines it uses. They include:

1. *input*() which returns the next input character;
2. *output*(c) which writes the character *c* on the output;
3. *unput*(c) which pushes the character *c* back onto the input stream to be read later by *input*().

By default, these routines are provided as macro definitions, but the user can override them and supply private versions. These routines define the relationship between external files and internal characters, and must all be retained or modified consistently. They may be redefined, to cause input or output to be transmitted to or from strange places, including other programs or internal memory; but the character set used must be consistent in all routines; a value of zero returned by *input*() must mean end-of-file; and the relationship between *unput*() and *input*() must be retained, or the lookahead will not work. **lex** does not look ahead at all if it does not have to, but every rule containing a slash (/) or ending in one of the following characters implies lookahead:

```
+ * ? $
```

Lookahead is also necessary to match an expression that is a prefix of another expression. See below for a discussion of the character set used by **lex**. The standard **lex** library imposes a 100 character limit on backup.

Another **lex** library routine that you sometimes want to redefine is *yywrap*() which is called whenever **lex** reaches an end-of-file. If *yywrap*() returns a 1, **lex** continues with the normal wrapup on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap*() that arranges for new input and returns 0. This instructs **lex** to continue processing. The default *yywrap*() always returns 1.

This routine is also a convenient place to print tables, summaries, etc. at the end of a program. Note that it is not possible to write a normal rule that recognizes end-of-file; the only access to this condition is through *yywrap*(). In fact, unless a private version of *input*() is supplied, a file containing nulls cannot be handled, since a value of 0 returned by *input*() is taken to be end-of-file.

5.13 Handling Ambiguous Source Rules

lex can handle ambiguous specifications. When more than one expression can match the current input, **lex** chooses the following:

- The longest match is preferred.
- Among rules that match the same number of characters, the first given rule is preferred.

For example, suppose the following rules are given:

```
integer       keyword action ...;
[a-z]+ identifier action ...;
```

If the input is *integers*, it is taken as an identifier, because:

```
[a–z]+
```

matches 8 characters while:

```
integer
```

matches only 7. If the input is *integer*, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g., *int*) does not match the expression *integer*, so the identifier interpretation is used.

The principle of preferring the longest match makes certain constructions dangerous, such as the following:

```
.*
```

For example:

```
'.*'
```

might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote.

Presented with the input:

```
'first' quoted string here, 'second' here
```

the above expression matches:

```
'first' quoted string here, 'second'
```

which is probably not what was wanted. A better rule to follow is the form:

```
'[^'\n]*'
```

which, on the above input, stops after 'first'. The consequences of errors like this are mitigated by the fact that the dot (.) operator does not match a newline. Therefore, no more than one line is ever matched by such expressions. Do not try to defeat this with expressions like:

```
[.\n]+
```

or their equivalents: the **lex** generated program will try to read the entire input file, causing internal buffer overflows.

Note that **lex** is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both *she* and *he* in an input text. Some of the **lex** rules to do this might be:

```
she     s++;
he      h++;
\n      |
.       ;
```

where the last two rules ignore everything besides *he* and *she*. Remember that the period (.) does not include the newline. Since *she* includes *he*, **lex** will normally not recognize the instances of *he* included in *she*, since once it has passed a *she*, those characters are gone.

Sometimes the user would like to override this choice. The action REJECT means go do the next alternative. It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of *he*. The following can then be applied:

```
she    {s++; REJECT;}
he     {h++; REJECT;}
\n     |
.      ;
```

These rules are one way of changing the previous example to do just that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, the user could note that *she* includes *he*, but not vice versa, and omit the REJECT action on *he*; in other cases, it would not be possible to tell which input characters were in both classes.

Consider the two rules:

```
a[bc]+ {... ; REJECT;}
a[cd]+ {... ; REJECT;}
```

If the input is *ab*, only the first rule matches, and on *ad*, only the second matches. The input string *accb* matches the first rule for four characters and then the second rule for three characters. In contrast, the input *accd* agrees with the second rule for four characters and then the first rule for three.

In general, REJECT is useful whenever the purpose of **lex** is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. Suppose a digram table of the input is desired; normally the digrams overlap, that is the word "the" is considered to contain both *th* and *he*. Assuming a two-dimensional array named *digram* is to be incremented, the appropriate source is as follows:

```
%%
[a-z][a-z]     {digram[yytext[0]][yytext[1]]++; REJECT;}
.      ;
\n     ;
```

where the REJECT is necessary to pick up a letter pair beginning at every character, rather than at every other character.

Remember that REJECT does not rescan the input. Instead, it remembers the results of the previous scan. This means that if a rule with trailing context is found, and REJECT executed, you must not have used *unput*() to change the incoming characters from the input stream. This is the only restriction placed on the ability to manipulate the not-yet-processed input.

5.14 Specifying Left Context Sensitivity

Sometimes it is desirable to have several sets of lexical rules applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The caret (^) operator is a prior context operator, recognizing immediately preceding left context just as the dollar sign ($) recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes three means of dealing with different environments:

1. The use of flags, when only a few rules change from one environment to another.

2. The use of start conditions with rules.

3. The use of multiple lexical analyzers running together.

In each case, there are rules that recognize the need to change the environment in which the following input text is analyzed, and to set some parameter in order to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since **lex** is not involved at all. It may be more convenient, however, to have **lex** remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when **lex** is in that start condition. The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are not similar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: copy the input to the output, changing the word *magic* to *first* on every line that began with the letter *a*; changing *magic* to *second* on every line that began with the letter *b*; and changing *magic* to *third* on every line that began with the letter *c*. All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with the following flag:

```
        int flag;
%%
^a      {flag='a'; ECHO;}
^b      {flag='b'; ECHO;}
^c      {flag='c'; ECHO;}
\n      {flag= 0 ; ECHO;}
magic   {
        switch (flag)
        {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
        case 'c': printf("third"); break;
        default: ECHO; break;
        }
        }
```

should be adequate.

To handle the same problem with start conditions, each start condition must be introduced to **lex** in the definitions section with a line reading:

```
%Start      name1 name2 ...
```

where the conditions may be named in any order. The word *Start* may be abbreviated to *s* or *S*. The conditions may be referenced at the head of a rule with angle brackets. For example:

```
<name1> expression
```

is a rule that is only recognized when **lex** is in the start condition *name1*. To enter a start condition, execute the action statement:

```
BEGIN name1;
```

which changes the start condition to *name1*. To return to the initial state, enter:

```
BEGIN 0;
```

which resets the initial condition of the **lex** automaton interpreter. A rule may be active in several start conditions; for example:

```
<name1,name2,name3>
```

is a legal prefix. Any rule not beginning with the <> prefix operator is always active.

The same example as before can also be written as:

```
%START AA BB CC
%%
^a      {ECHO; BEGIN AA;}
^b      {ECHO; BEGIN BB;}
^c      {ECHO; BEGIN CC;}
\n      {ECHO; BEGIN 0;}
<AA>magic       printf("first");
<BB>magic printf("second");
<CC>magic printf("third");
```

where the logic is exactly the same as in the previous method of handling the problem, but **lex** does the work rather than the user's code.

5.15 Specifying Source Definitions

Remember the format of the **lex** source:

```
{definitions}
%%
{rules}
%%
{user routines}
```

So far, only the rules have been described. You will need additional options, though, to define variables for use in your program and for use by **lex**. These can go either in the definitions section or in the rules section.

Remember that **lex** is turning the rules into a program. Any source not intercepted by **lex** is copied into the generated program. There are three classes of such things:

1. Any line that is not part of a **lex** rule or action which begins with a blank or tab is copied into the **lex** generated program. Such source input prior to the first %% delimiter will be external to any function in the code; if it appears immediately after the first %%, it appears in an appropriate place for declarations in the function written by **lex** which contains the actions. This material must look like program fragments, and should precede the first **lex** rule.

   As a side effect of the above, lines that begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be used to include comments in either the **lex** source or the generated code. The comments should follow C language conventions.

2. Anything included between lines containing only %{ and %} is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3. Anything after the third %% delimiter, regardless of format, is copied out after the **lex** output.

Definitions intended for **lex** are given before the first %% delimiter. Any line in this section not contained between %{ and %}, and beginning in column 1, is assumed to define **lex** substitution strings. The format of such lines is:

```
name translation
```

and it causes the string given as a translation to be associated with the name. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the {name} syntax in a rule. For example, using {D} for the digits and {E} for an exponent field might abbreviate rules to recognize numbers:

```
D                       [0-9]
E                       [DEde][-+]?{D}+
%%
{D}+                    printf("integer");
{D}+"."{D}*({E})?       |
{D}*"."{D}+({E})?       |
{D}+{E}                 printf("real");
```

Note the first two rules for real numbers; both require a decimal point and contain an optional exponent field, but the first requires at least one digit before the decimal point and the second requires at least one digit after the decimal point. To correctly handle the problem posed by a FORTRAN expression such as: *35.EQ.I*, which does not contain a real number, a context-sensitive rule such as

```
[0-9]+/"."EQ printf("integer");
```

could be used in addition to the normal rule for integers.

The definitions section may also contain other commands, including a character set table, a list of start conditions, or adjustments to the default size of arrays within **lex** itself for larger source programs. These possibilities are discussed in the section "Source Format".

5.16 lex and yacc

If you want to use **lex** with **yacc**, note that what **lex** writes is a program named *yylex*(), the name required by **yacc** for its analyzer. Normally, the default main program on the **lex** library calls this routine, but if **yacc** is loaded, and its main program is used, **yacc** will call *yylex*(). In this case, each **lex** rule should end with:

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to **yacc**'s names for tokens is to compile the **lex** output file as part of the **yacc** output file by placing the line:

```
# include "lex.yy.c"
```

in the last section of **yacc** input. Supposing the grammar to be named *good*, and the lexical rules to be named *better*, the XENIX command sequence can be entered as:

```
yacc good
lex better
cc y.tab.c -ly -ll
```

The **yacc** library (-ly) should be loaded before the **lex** library, to obtain a main program which invokes the **yacc** parser. The generation of **lex** and **yacc** programs can be done in either order.

As a trivial problem, consider copying an input file while adding 3 to every positive number divisible by 7. Here is a suitable **lex** source program to do just that:

```
%%
        int k;
[0-9]+ {
                k = atoi(yytext);
                if (k%7 == 0)
                  printf("%d", k+3);
                else
                  printf("%d",k);
                }
```

The rule [0–9]+ recognizes strings of digits; **atoi** (see **atof**(S)) converts the digits to binary and stores the result in *k*. The remainder operator (%) is used to check whether *k* is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as 49.63 or X7.

Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as shown below:

```
%%
        int k;
-?[0-9]+        {
                k = atoi(yytext);
                printf("%d", k%7 == 0 ? k+3 : k);
                }
-?[0-9.]+               ECHO;
[A-Za-z][A-Za-z0-9]+    ECHO;
```

Numerical strings containing a decimal point or preceded by a letter will be picked up by one of the last two rules, and not changed. The **if–else** has been replaced by a C conditional expression to save space; the form *a?b:c* means: if *a* then *b* else *c*.

For an example of statistics gathering, the following is a program which makes histograms of word lengths, where a word is defined as a string of letters.

```
        int lengs[100];
%%
[a-z]+  lengs[yyleng]++;
.       |
\n      ;
%%
yywrap()
{
int i;
printf("Length No. words\n");
for(i=0; i<100; i++)
  if (lengs[i] > 0)
    printf("%5d%10d\n",i,lengs[i]);
return(1);
}
```

This program accumulates the histogram, while producing no output. At the end of the input it prints the table. The final statement *return*(1); indicates that **lex** is to perform wrapup. If *yywrap*() returns zero (false) it implies that further input is available and the program is to continue reading and processing. To provide a *yywrap*() that never returns true causes an infinite loop.

As a larger example, here are some parts of a program written to convert double precision FORTRAN to single precision FORTRAN. Because FORTRAN does not distinguish between upper- and lowercase letters, this routine begins by defining a set of classes including both cases of each letter:

```
a       [aA]
b       [bB]
c       [cC]
.        .
.        .
.        .
z       [zZ]
```

An additional class recognizes white space:

```
W       [ \t]*
```

The first rule changes *double precision* to *real*, or *DOUBLE PRECISION* to *REAL*.

```
{d}{o}{u}{b}{l}{e}{W}{p}{r}{e}{c}{i}{s}{i}{o}{n} {
  printf(yytext[0]=='d'? "real" : "REAL");
  }
```

Care is taken throughout this program to preserve the case of the original program. The conditional operator is used to select the proper form of the keyword. The next rule copies continuation card indications to avoid confusing them with constants:

```
^"     "[^ 0]       ECHO;
```

In the regular expression, the quotes surround the blanks. It is interpreted as beginning of line, then five blanks, then anything but blank or zero." Note the two different meanings of the caret (^) here. The following is a **lex** program that changes double precision constants to ordinary floating constants.

```
[0-9]+{W}{d}{W}[+-]?{W}[0-9]+                              |
[0-9]+{W}"."{W}{d}{W}[+-]?{W}[0-9]+             |
"."{W}[0-9]+{W}{d}{W}[+-]?{W}[0-9]+             {
  /* convert constants */
  for(p=yytext; *p != 0; p++)
    {
    if (*p== 'd' ||*p == 'D')
      *p+= 'e'- 'd';
      ECHO;
    }
```

After the floating point constant is recognized, it is scanned by the **for** loop, to find the letter "d" or "D". The program then adds "'e'-'d'", which converts it to the next letter of the alphabet. The modified constant, now single precision, is written out again. The following is a series of names which must be respelled, in order to remove their initial "d". By using the array *yytext,* the same action suffices for all the names (only a sample of a rather long list is given here).

```
{d}{s}{i}{n}          |
{d}{c}{o}{s}          |
{d}{s}{q}{r}{t}       |
{d}{a}{t}{a}{n}       |
...
{d}{f}{l}{o}{a}{t}printf("%s",yytext+1);
```

Another list of names must have the initial *d* changed to initial *a*:

```
{d}{l}{o}{g}          |
{d}{l}{o}{g}10        |
{d}{m}{i}{n}1         |
{d}{m}{a}{x}1         {
        yytext[0]+='a'-'d';
        ECHO;
        }
```

And one routine must have the initial *d* changed to initial *r*:

```
{d}1{m}{a}{c}{h} {
        yytext[0]+='r' -'d';
        ECHO;
}
```

To avoid such names as *dsinx* being detected as instances of *dsin,* some final rules pick up longer words as identifiers and copy some surviving characters:

```
[A-Za-z][A-Za-z0-9]*    |
[0-9]+                  |
\n                      |
.       ECHO;
```

Note that this program is not complete; it does not deal with the spacing problems in FORTRAN or with the use of keywords as identifiers.

5.17 Specifying Character Sets

The programs generated by **lex** handle character I/O only through the *input*(), *output*(), and *unput*() routines. Thus, the character representation provided in these routines is accepted by **lex** and employed to return values in *yytext*. For internal use, a character is represented as a small integer, and, if the standard library is used, it has a value equal to the integer value of the bit pattern representing the character on the host computer. Normally, the letter *a* is represented in the same form as the character constant:

```
'a'
```

If this interpretation is changed, by providing I/O routines that translate the characters, **lex** must be told about it, by giving a translation table. This table must be in the definitions section, and must be bracketed by lines containing only *%T*. The table contains lines of the following form:

```
{integer} {character string}
```

which indicate the value associated with each character. For example:

```
%T
1     Aa
2     Bb
...
26    Zz
27    \n
28    +
29    -
30    0
31    1
...
39    9
%T
```

This table maps the lowercase and uppercase letters together into the integers 1 through 26, the newline into 27, the plus (+) and minus (−) into 28 and 29, and the digits into 30 through 39. Note the escape for a newline. If a table is supplied, every character that is to appear either in the rules or in any valid input must be included in the table. No character may be assigned the number 0, and no character may be assigned a larger number than the size of the hardware character set.

5.18 Source Format

The general form of a **lex** source file is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

The definitions section contains a combination of:

1. Definitions, in the form "name space translation".

2. Included code, in the form "space code".

3. Included code, in the form:

   ```
   %{
   code
   %}
   ```

4. Start conditions, given in the form:

   ```
   %S name1 name2 ...
   ```

5. Character set tables, in the form:

   ```
   %T
   number space character-string
   %T
   ```

6. Changes to internal array sizes, in the form:

   ```
   %x   nnn
   ```

   where *nnn* is a decimal integer representing an array size and *x* selects the parameter as follows:

Letter	Parameter
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the rules section have the form:

expression action

where the action may be continued on succeeding lines by using braces to delimit it.

Regular expressions in **lex** use the following operators:

x	The character "x".
"x"	An "x", even if x is an operator.
\x	An "x", even if x is an operator.
[xy]	The character x or y.
[x–z]	The characters x, y or z.
[^x]	Any character but x.
.	Any character but newline.
^x	An x at the beginning of a line.
<y>x	An x when **lex** is in start condition y.
x$	An x at the end of a line.
x?	An optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	An x or a y.
(x)	An x.
x/y	An x, but only if followed by y.
{xx}	The translation of xx from the definitions section.
x{m,n}	*m* through *n* occurrences of x.

Chapter 6

yacc: A Compiler-Compiler

6.1 Introduction 6-1

6.2 Specifications 6-4

6.3 Actions 6-6

6.4 Lexical Analysis 6-9

6.5 How the Parser Works 6-11

6.6 Ambiguity and Conflicts 6-16

6.7 Precedence 6-21

6.8 Error Handling 6-23

6.9 The yacc Environment 6-26

6.10 Preparing Specifications 6-27

6.11 Input Style 6-27

6.12 Left Recursion 6-28

6.13 Lexical Tie-ins 6-29

6.14 Handling Reserved Words 6-30

6.15 Simulating Error and Accept in Actions 6-30

6.16 Accessing Values in Enclosing Rules 6-30

6.17 Supporting Arbitrary Value Types 6-31

6.18 A Small Desk Calculator 6-32

6.19 yacc Input Syntax 6-35

6.20 An Advanced Example 6-38

6.1 Introduction

Computer program input generally has some structure; every computer program that accepts input can be thought of as defining an input language which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often lax about checking their inputs for validity.

yacc(CP) provides a general tool for describing the input to a computer program. The name **yacc** stands for "yet another compiler-compiler". The **yacc** user specifies the structures of his input, together with the code to be invoked as each structure is recognized. **yacc** turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have flow control in the user's application handled by this subroutine.

The input subroutine produced by **yacc** calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle peculiar features such as comment and continuation conventions, which typically defy easy grammatical specification. The class of specifications accepted is a very general one: LALR grammars with certain rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., **yacc** has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a FORTRAN debugging system.

yacc provides a general tool for imposing structure on the input to a computer program. The **yacc** user prepares a specification of the input process; this includes rules describing the input structure, the code to be invoked when these rules are recognized, and a low-level routine to do the basic input. **yacc** then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (called the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules. When one of these rules has been recognized, user code supplied for this rule is invoked. Note that actions have the ability to return values and make use of the values of other actions.

yacc is written in a portable dialect of C and the actions, and output subroutine, are also written in C. Moreover, many of the syntactic conventions of **yacc** follow the C language syntax.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be:

```
        date : month_name day ',' year  ;
```

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma (,) is enclosed in single quotation marks; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input:

```
        July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. A structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules:

```
        month_name : 'J' 'a' 'n' ;
        month_name : 'F' 'e' 'b' ;
            .
            .
            .
        month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond **yacc**'s ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a *month_name* was seen; in this case, *month_name* would be a token.

Literal characters, such as the comma, must also be passed through the lexical analyzer and are considered tokens.

Specification files are very flexible. It is relatively easy to add the following rule to the example shown above:

```
date : month '/' day '/' year ;
```

allowing:

```
7/4/1776
```

as a synonym for:

```
July 4, 1776
```

In most cases, this new rule could be slipped in to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, **yacc** fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to **yacc**. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While **yacc** cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for **yacc** to handle are also frequently difficult for you to handle. Some users have reported that the discipline of formulating valid **yacc** specifications for their input revealed errors of conception or design early in the program development stage.

The next several sections describe:

- The preparation of grammar rules.
- The preparation of the user supplied actions associated with the grammar rules.
- The preparation of lexical analyzers.
- The operation of the parser.
- Various reasons why **yacc** may be unable to produce a parser from a specification, and what to do about it.

- A simple mechanism for handling operator precedences in arithmetic expressions.
- Error detection and recovery.
- The operating environment and special features of the parsers **yacc** produces.
- Gives some suggestions which should improve the style and efficiency of the specifications.

6.2 Specifications

Names refer to either tokens or nonterminal symbols. **yacc** requires token names to be declared as such. In addition, for reasons discussed later, it is often desirable to include the lexical analyzer as part of the specification file. It may be useful to include other programs as well. Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent %% marks. (The percent sign (%) is generally used in **yacc** specifications as an escape character.)

In other words, a full specification file is shown as:

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the program section is omitted, the second %% mark may be omitted also; thus, the smallest legal **yacc** specification is:

```
%%
rules
```

Blanks, tabs, and newlines are ignored; note that they may not appear in names or multicharacter reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* ... */, as in C.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and *BODY* represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Names may be of arbitrary length, and may be made up of letters, dot (.), the underscore (_), and noninitial digits. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotation marks ('). As in C, the backslash (\) is an escape character within literals, and all the C escapes are recognized. Thus:

'\n'	Newline
'\r'	Return
'\''	Single quotation mark
'\\'	Backslash
'\t'	Tab
'\b'	Backspace
'\f'	Form feed
'\xxx'	"xxx" in octal

For a number of technical reasons, the ASCII NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, then the vertical bar (|) can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules:

```
A : B C D ;
A : E F ;
A : G ;
```

can be given to **yacc** as:

```
A : B C D
 | E F
 | G
 ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, it can be indicated as follows:

```
empty : ;
```

Names representing tokens must be declared; this can be done by entering:

```
%token name1 name2 ...
```

in the declarations section. (See Sections 3 , 5, and 6 in this chapter for a detailed explanation.) Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start symbol
```

The end of the parser input is signaled by a special token, called the endmarker. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it then accepts the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 6.3, below. Usually, the endmarker represents some reasonably obvious I/O status, such as the end of a file or the end of a record.

6.3 Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement; it can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces { and }. For example:

```
A : '(' B ')'
    {   hello( 1, "abc"); }
```

and:

```
XXX : YYY ZZZ
      { printf("a message\n");
       flag = 25;}
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The dollar sign ($) is used as a signal to **yacc** in this context.

To return a value, the action normally sets the pseudo-variable *$$* to some value. For example, an action that does nothing but return the value 1 is:

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, ..., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is:

```
A : B C D ;
```

then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule:

```
expr : '(' expr ')' ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by:

```
expr : '(' expr ')' { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form:

```
A : B ;
```

do not need to have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. **yacc** permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the $ mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left.

Thus, in the rule:

```
A : B
  { $$ = 1; }
  C
  { x=$2; y=$3; }
  ;
```

the effect is to set *x* to 1, and to set *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by **yacc** by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. **yacc** actually treats the above example as if it had been written:

```
$ACT : /* empty */
        { $$ = 1; }
  ;

A  : B $ACT C
        { x=$2; y=$3; }
  ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call:

```
node( L, n1, n2 )
```

creates a node with label L, with the descendants n1 and n2, and returns the index of the newly created node. Then, a parse tree can be built by supplying actions such as:

```
expr : expr '+' expr
        { $$ = node( '+', $1, $3 ); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks %{ and %}. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example:

```
        %{ int variable = 0; %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The **yacc** parser uses only names beginning in *yy*; the user should avoid such names.

In the examples shown, all the values are integers. A discussion of values of other types are found in a later section.

6.4 Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex()*. The function returns an integer, called the token number, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval.*

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by **yacc**, or chosen by the user. In either case, the *#define* mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name *DIGIT* has been defined in the declarations section of the **yacc** specification file. The relevant portion of the lexical analyzer might look like the example on the following page.

```
yylex(){
    extern int yylval;
    int c;
    ...
    c = getchar();
    ...
    switch( c ) {
        ...
    case '0':
    case '1':
     ...
    case '9':
        yylval = c-'0';
        return( DIGIT );
        ...
        }
    ...
```

The intent is to return a token number of *DIGIT*, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the program section of the specification file, the identifier, *DIGIT*, is defined as the token number associated with the token *DIGIT*.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of the token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error handling, and should not be used naively.

As mentioned above, the token numbers may be chosen by **yacc** or by the user. In the default situation, the numbers are chosen by **yacc**. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user. Hence, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is **lex**, discussed in a previous section. These lexical analyzers are designed to work in close harmony with **yacc** parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. **lex** can be easily used to produce quite complicated lexical analyzers, but there are some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

6.5 How the Parser Works

yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more understandable.

The parser produced by **yacc** consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and the lookahead token has not been read.

The machine has four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex*() to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF  shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not. In fact, the default action (represented by a .) is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action:

```
	.   reduce 18
```

refers to grammar rule 18, while the action:

```
	IF   shift 34
```

refers to state 34.

Suppose the rule being reduced is shown on the following page.

```
	A : x y z ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped is equal to the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a goto action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
	A   goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action turns back the clock in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side of the rule at that time. If the right hand side of the rule is empty, no states are popped off of the stack; the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable, *yylval,* is copied onto the value stack. After return from the user code, the reduction is carried out. When the goto action is done, the external variable, *yyval,* is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing; the error recovery (as opposed to the detection of error) will be discussed in a later section.

Consider the following example:

```
%token DING DONG DELL
%%
rhyme : sound place
   ;
sound : DING DONG
   ;
place : DELL
   ;
```

When **yacc** is invoked with the **−v** option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is given on the next page.

```
state 0
    $accept : _rhyme $end

    DING shift 3
    . error

    rhyme goto 1
    sound goto 2

state 1
    $accept : rhyme_$end

    $end accept
    . error

state 2
    rhyme : sound_place

    DELL shift 5
    . error

    place goto 4


state 3
    sound : DING_DONG

    DONG shift 6
    . error

state 4
    rhyme : sound place_ (1)

    . reduce 1

state 5
    place : DELL_ (3)

    . reduce 3

state 6
    sound : DING DONG_ (2)

    . reduce 2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The underscore character (_) is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is:

```
        DING DONG DELL
```

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is *shift 3*, so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is *shift 6*, so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
        sound : DING DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. It then checks the description of state 0, looking for a goto on *sound*, as shown below:

```
        sound goto 2
```

is obtained; meaning state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is *shift 5*, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme*, causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained; indicated by a *$end* in the file. When the endmarker is accepted, the action is called state 1. This successfully ends the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG, DING DONG, DING DONG DELL DELL*, etc. A few minutes spent with this and the other simple examples will probably be repaid when problems arise in more complicated contexts.

6.6 Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule:

```
expr : expr '-' expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is:

```
expr - expr - expr
```

the rule allows this input to be structured as either:

```
( expr - expr ) - expr
```

or as:

```
expr - ( expr - expr )
```

(The first is called left association, the second is called right association).

yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given input such as:

```
expr - expr - expr
```

When the parser has read the second expr, the input that has been read:

```
expr - expr
```

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule. After applying this rule, the input is reduced to *expr* (the left side of the rule). The parser would then read the final part of the input:

```
- expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has read:

```
expr - expr
```

it could defer the immediate application of the rule, and continue reading the input until it comes across:

```
expr - expr - expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving:

```
expr - expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read:

```
expr - expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a shift/reduce conflict. It may also happen that the parser has a choice of two legal reductions; this is called a reduce/reduce conflict. Note that there are never any shift/shift conflicts.

When there are shift/reduce or reduce/reduce conflicts, **yacc** still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a disambiguating rule.

yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.

2. In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than **yacc** can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, **yacc** always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read, but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, **yacc** will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an if-then-else construction:

```
stat : IF '(' cond ')' stat
  | IF '(' cond ')' stat ELSE stat
  ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the simple-if rule, and the second the if-else rule.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 ) {
    IF ( C2 ) S1
    }
ELSE S2
```

or

```
IF ( C1 ) {
    IF ( C2 ) S1
    ELSE S2
    }
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last *IF* immediately preceding the *ELSE*. In this example, consider the situation where the parser has seen:

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get:

```
IF ( C1 ) stat
```

and then read the remaining input:

```
ELSE S2
```

and reduce:

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to groupings of the above listed input.

On the other hand, the *ELSE* may be shifted, *S2* read, and the right hand portion of:

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

to be reduced by the if-else rule to get:

```
IF ( C1 ) stat
```

which can then be reduced by the simple-if rule. This leads to the second grouping of the previously listed input, which is usually desired.

Once again, the parser can do two valid things – there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs that have already been seen, such as:

```
IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of **yacc** are best understood by examining the verbose (**−v**) option output file. The output corresponding to the above conflict state might be like the example shown on the next page.

```
23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

    stat : IF ( cond ) stat_   (18)
    stat : IF ( cond ) stat_ELSE stat

    ELSE  shift 45
    .   reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which have been read. Thus in the example, in state 23 the parser has read input corresponding to:

```
IF ( cond ) stat
```

and the two grammar rules shown, are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line:

```
stat : IF ( cond ) stat ELSE_stat
```

Note that the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by ".", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

```
stat : IF '(' cond ')' stat
```

Once again, notice that the numbers following shift commands refer to other states, while the numbers following reduce commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules that can be reduced. In most one states, there will be at most 1 reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the references previously shown can be checked; or the services of a knowledgeable user can be requested.

6.7 Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form:

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence of all the operators, and the associativity of the binary operators. This information is sufficient to allow **yacc** to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a **yacc** keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence. Thus:

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in FORTRAN. Thus:

```
A .LT. B .LT. C
```

is illegal in FORTRAN, and such an operator would be described with the keyword %nonassoc in **yacc**.

As an example of the behavior of these declarations, the description:

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr : expr '=' expr
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | NAME
    ;
```

might be used to structure the input:

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but have different precedences. An example is unary and binary '–'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. The %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '-'
%left '*' '/'

%%

expr : expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | '-' expr %prec '*'
    | NAME
    ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by the %token, as well.

The precedences and associativities are used by **yacc** to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.

2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.

4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by **yacc**. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially cookbook fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

6.8 Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found. It is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser restarted after an error. A general class of algorithms to perform this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, **yacc** provides a simple, but reasonably general feature. The token name *error* is reserved for error handling. This name can be used in grammar rules; in effect, it suggests locations where errors are expected, and where recovery might take place. The parser pops its stack until it enters a state where the token *error* is legal. It then behaves as if the token *error* were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in an error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, the following rule:

```
stat : error
```

would, in effect, mean that on a syntax error, the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error when there actually is no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as:

```
stat : error ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ´;´. All tokens after the error and before the next ´;´ cannot be shifted, and are discarded. When the ´;´ is seen, this rule will be reduced, and any cleanup action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be:

```
input : error '\n' { printf( "Reenter line: "); } input
        { $$ = $4;}
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written:

```
input : error '\n'
        { yyerrok;
         printf( "Reenter last line: "); }
        input
        { $$ = $4; }
        ;
```

As mentioned above, the token seen immediately after the *error* symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like:

```
stat : error
      { resynch();
       yyerrok ;
       yyclearin ; }
      ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors. Moreover, the user can get control to deal with the error actions required by other portions of the program.

6.9 The yacc Environment

When the user inputs a specification to **yacc**, the output is a file of C program, called *y.tab.c* on most systems. The function produced by **yacc** is called *yyparse*()*;* it is an integer valued function. When it is called, it repeatedly calls *yylex*, the lexical analyzer supplied by the user to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse*() returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse*() returns the value 0.

The user must provide a certain amount of environment for this parser, in order to obtain a working program. For example, as with every C program, a program called *main*() must be defined, that eventually calls *yyparse*() In addition, a routine called *yyerror*() prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using **yacc**, a library has been provided with default versions of *main*() and *yyerror*(). The name of this library is system dependent; on many systems the library is accessed by a **−ly** argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
   return( yyparse() );
   }
```

and

```
#include <stdio.h>

yyerror(s) char *s; {
   fprintf( stderr, "%s\n", s );
   }
```

The argument to *yyerror*() is a string containing an error message, usually the string *syntax error*. The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main*() program is probably supplied by the user (to read arguments, etc.) the **yacc** library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

6.10 Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

6.11 Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. Thus, try to be consistent with the following conventions when entering a specification file:

1. Use uppercase letters for token names, lowercase letters for nonterminal names. This rule helps you to know who to blame when things go wrong.

2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

3. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

4. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

5. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The examples in the text of this section follow this style (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the scope of action code.

6.12 Left Recursion

The algorithm used by the **yacc** parser encourages so called left recursive grammar rules of the form:

```
name : name rest_of_rule ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list : item
  | list ',' item
  ;
```

and

```
seq : item
  | seq item
  ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as:

```
seq : item
  | item seq
  ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion whenever possible.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq : /* empty */
  | seq item
  ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if **yacc** is asked to decide which empty sequence it has seen, when it has not seen enough to know!

6.13 Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
 int dflag;
%}
 ... other declarations ...

%%

prog  : decls stats
    ;

decls : /* empty */
        {   dflag = 1; }
    | decls declaration
    ;

stats : /* empty */
        {   dflag = 0; }
    | stats statement
    ;

  ... other rules ...
```

The flag **dflag** is now 0 when reading statements, and 1, when reading declarations, except for the first token in the first statement. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of back door approach can be over done. Nevertheless, it represents a way of doing some things that are difficult to do otherwise.

6.14 Handling Reserved Words

Some programming languages permit the user to use words like *if*, which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of **yacc**; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance of "if" is a variable". The user can try this, but it is difficult. It is best that keywords be reserved; that is, be forbidden for use as variable names.

6.15 Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros *YYACCEPT* and *YYERROR*. *YYACCEPT* causes *yyparse*() to return the value 0; *YYERROR* causes the parser to behave as if the current input symbol had been a syntax error; *yyerror*() is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

6.16 Accessing Values in Enclosing Rules

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case, the digit may be 0 or negative. Consider:

```
sent  : adj noun verb adj noun
        { look at the sentence ... }
      ;



adj   : THE  { $$ = THE; }
      | YOUNG { $$ = YOUNG; }
      ...
      ;

noun  : DOG  { $$ = DOG; }
      | CRONE { if( $0 == YOUNG ){
              printf( "what?\n" );
              }
          $$ = CRONE;
        }
      ;
      ...
```

In the action following the word *CRONE*, a check is made that the preceding token shifted was not *YOUNG*. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times, this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

6.17 Supporting Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. **yacc** can also support values of other types, including structures. In addition, **yacc** keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The **yacc** value stack is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, **yacc** will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as **lint**(C), will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where **yacc** cannot easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union {
   body of union ...
   }
```

This declares the **yacc** value stack, and the external variables *yylval* and *yyval*, to have type equal to this union. If **yacc** was invoked with the **−d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable *YYSTYPE* to represent this union. Thus, the header file might also have said:

```
typedef union {
    body of union ...
    } YYSTYPE;
```

The header file must be included in the declaration section, by use of %{ and %}.

Once *YYSTYPE* is defined, the union member names must be associated with the various terminal and nonterminal names. The construction:

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords, %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus:

```
%left <optype> '+' '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say:

```
%type <nodetype> expr stat
```

A couple of cases remain where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no predefined type. Similarly, reference to left context values (such as $0 – see the previous subsection) leaves **yacc** with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is:

```
rule : aaa { $<intval>$ = 3; } bbb
       { fun( $<intval>2, $<other>0 ); }
   ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in a later section. The facilities in this subsection are not triggered until they are used. In particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the **yacc** value stack is used to hold *int*'s, as was true historically.

6.18 A Small Desk Calculator

The following example shows the complete **yacc** specification for a small desk calculator. The desk calculator has 26 registers, labeled *a* through *z*, and accepts arithmetic expressions made up of the operators +, –, *, /, % (mod operator), & (bitwise and), | (bitwise or), and = (assignment). If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a **yacc** specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules. This job is better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS /* precedence for unary minus */

%%   /* beginning of rules section */

list : /* empty */
   | list stat '\n'
   | list error '\n'
      { yyerrok; }
   ;

stat : expr
      { printf("%d\n", $1); }
   |  LETTER '=' expr
        { regs[$1] = $3; }
   ;
```

```
expr :'('expr')'
       {$$=$2;}
    |expr'+'expr
       {$$=$1+$3;}
    |expr'-'expr
       {$$=$1-$3;}
    |expr'*'expr
       {$$=$1*$3;}
    |expr'/'expr
       {$$=$1/$3;}
    |expr'%'expr
       {$$=$1%$3;}
    |expr'&'expr
       {$$=$1&$3;}
    |expr'|'expr
       {$$=$1|$3;}
    |'-'expr %prec UMINUS
       {$$=-$2;}
    |LETTER
       {$$=regs[$1];}
    |number
    ;


         number :DIGIT
                {$$=$1;base=($1==0)?8:10;}
            |number DIGIT
                {$$=base*$1+$2;}
            ;

         %%    /* start of programs */
```

```
yylex() {        /* lexical analysis routine */
                 /* returns LETTER for a lowercase letter, */
                 /* yylval = 0 through 25 */
                 /* return DIGIT for a digit, */
                 /* yylval = 0 through 9 */
                 /* all other characters */
                 /* are returned immediately */

    int c;

    while( (c=getchar()) == ' ') { /* skip blanks */ }

                 /* c is now nonblank */

    if( islower( c ) ) {
        yylval = c - 'a';
        return ( LETTER );
        }
    if( isdigit( c ) ) {
        yylval = c - '0';
        return( DIGIT );
        }
    return( c );
    }
```

6.19 yacc Input Syntax

This section has a description of the **yacc** input syntax, as a **yacc** specification. Context dependencies, etc., are not considered. Ironically, the **yacc** input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise, it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decides whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token *C_IDENTIFIER*. Otherwise, it returns *IDENTIFIER*. Literals (quoted strings) are also returned as *IDENTIFIER*, but never as part of a *C_IDENTIFIER*.

```
    /* grammar for the input to yacc */

   /* basic entities */
%token IDENTIFIER      /* includes identifiers and literals */
%token C_IDENTIFIER /* identifier followed by colon    */
%token NUMBER          /* [0-9]+  */

   /* reserved words: %type => TYPE, %left => LEFT, etc. */

%token LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token MARK  /* the %% mark */
%token LCURL  /* the %{ mark */
%token RCURL  /* the %} mark */

   /* ascii character literals stand for themselves */

%start spec

%%

spec  : defs MARK rules tail
    ;

tail  : MARK  { Eat up the rest of the file }
    | /* empty: the second MARK is optional */
    ;

defs  : /* empty */
    | defs def
    ;

def   : START IDENTIFIER
    | UNION { Copy union definition to output }
    | LCURL { Copy C code to output file } RCURL
    | ndefs rword tag nlist
    ;

rword  : TOKEN
    | LEFT
    | RIGHT
    | NONASSOC
    | TYPE
    ;
```

```
tag   : /* empty: union tag is optional */
    |´<´ IDENTIFIER ´>´
    ;

nlist  : nmno
    |nlist nmno
    |nlist ´,´ nmno
    ;

nmno  : IDENTIFIER       /* Literal illegal with % type */
    | IDENTIFIER NUMBER /* Illegal with % type */
    ;

    /* rules section */

rules  : C_IDENTIFIER rbody prec
    |rules rule
    ;

rule  : C_IDENTIFIER rbody prec
    |'|' rbody prec
    ;

rbody  : /* empty */
    |rbody IDENTIFIER
    |rbody act
    ;


act   : ´{´ { Copy action, translate $$, etc. } ´}´
    ;

prec  : /* empty */
    |PREC IDENTIFIER
    |PREC IDENTIFIER act
    |prec ´;´
    ;
```

6.20 An Advanced Example

This section shows an example of a grammar using some of the advanced features discussed in earlier sections. The desk calculator example is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, −, *, /, unary −, and = (assignment), and has 26 floating point variables, *a* through *z*. Moreover, it also understands intervals, written as:

```
(x,y)
```

where *x* is less than or equal to *y*. There are 26 interval-valued variables *A* through *Z* that may also be used. Assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of **yacc** and C. Intervals are represented by a structure, consisting of the left and right end-point values, stored as double precision values. This structure is given a type name, *INTERVAL*, by using **typedef**. The **yacc** value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of *YYERROR* to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of **yacc** is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g., scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through **yacc**: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

```
2.5 + (3.5 - 4.)
```

and

```
2.5 + (3.5, 4.)
```

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the comma (,) is read. By this time, 2.5 is finished, and the parser cannot go back and change. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is circumvented by having two rules for each binary interval-valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. However, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file. In this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine **atof**(S) is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and leading to error recovery.

```
%{

# include <stdio.h>
# include <ctype.h>

typedef struct interval {
   double lo, hi;
   } INTERVAL;

INTERVAL vmul(), vdiv();

double  atof();

double  dreg[26];
INTERVAL vreg[26];

%}

%start  lines

%union {
   int ival;
   double dval;
   INTERVAL vval;
   }

%token <ival> DREG VREG  /* indices into dreg, vreg arrays */

%token <dval> CONST    /* floating point constant */

%type <dval> dexp     /* expression */

%type <vval> vexp     /* interval expression */
```

```
    /* precedence information about the operators */

%left '+' '-'
%left '*' '/'
%left UMINUS    /* precedence for unary minus */

%%

lines : /* empty */
    | lines line
    ;

line  : dexp '\n'
        { printf("%15.8f\n", $1); }
    | vexp '\n'
        { printf("(%15.8f, %15.8f )\n", $1.lo, $1.hi); }
    | DREG '=' dexp '\n'
        { dreg[$1] = $3; }
    | VREG '=' vexp '\n'
        { vreg[$1] = $3; }
    | error '\n'
        { yyerrok; }
    ;

dexp  : CONST
    | DREG
        { $$ = dreg[$1]; }
    | dexp '+' dexp
        { $$ = $1 + $3; }
    | dexp '-' dexp
        { $$ = $1 - $3; }
    | dexp '*' dexp
        { $$ = $1 * $3; }
    | dexp '/' dexp
        { $$ = $1 / $3; }
    | '-' dexp %prec UMINUS
        { $$ = - $2; }
    | '(' dexp ')'
        { $$ = $2; }
    ;
```

```
vexp  : dexp
        { $$.hi = $$.lo = $1; }
    | '(' dexp ',' dexp ')'
        {
         $$.lo = $2;
         $$.hi = $4;
         if( $$.lo > $$.hi ){
             printf("interval out of order\n");
             YYERROR;
             }
           }
    | VREG
        { $$ = vreg[$1]; }
    | vexp '+' vexp
        { $$.hi = $1.hi + $3.hi;
          $$.lo = $1.lo + $3.lo; }
    | dexp '+' vexp
        { $$.hi = $1 + $3.hi;
          $$.lo = $1 + $3.lo; }
    | vexp '-' vexp
        { $$.hi = $1.hi - $3.lo;
          $$.lo = $1.lo - $3.hi; }
    | dexp '-' vexp
        { $$.hi = $1 - $3.lo;
          $$.lo = $1 - $3.hi;}
    | vexp '*' vexp
        { $$ = vmul( $1.lo, $1.hi, $3 ); }
    | dexp '*' vexp
        { $$ = vmul( $1, $1, $3 ); }
    | vexp '/' vexp
        { if ( dcheck( $3 ) ) YYERROR;
         $$ = vdiv( $1.lo, $1.hi, $3 ); }
    | dexp '/' vexp
        { if ( dcheck( $3 ) ) YYERROR;
         $$ = vdiv( $1, $1, $3 ); }
    | '-' vexp %prec UMINUS
        { $$.hi = -$2.lo; $$.lo = -$2.hi; }
    | '(' vexp ')'
        {   $$ = $2; }
    ;

%%

# define BSZ 50 /* buffer size for fp numbers */

    /* lexical analysis */
```

```
yylex(){
    register c;
                  { /* skip over blanks */ }
    while( ( c = getchar() ) == ' ')

    if ( isupper(c) ){
        yylval.ival = c - 'A';
        return( VREG );
        }
    if ( islower(c) ){
        yylval.ival = c - 'a';
        return( DREG );
        }

    if( isdigit( c ) || c=='.' ){
        /* gobble up digits, points, exponents */

        char buf[BSZ+1], *cp = buf;
        int dot = 0, exp = 0;

        for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

            *cp = c;
            if ( isdigit(c) ) continue;
            if ( c == '.' ) {
                if ( dot++ || exp ) return( '.' );
                                  /* above causes syntax error */
                continue;
                }

            if (c == 'e' ) {
                if ( exp++ ) return( 'e' );
                                  /* above causes syntax error */
                continue;
                }

            /* end of number */
            break;
            }
        *cp = '\0';
        if( (cp-buf) >= BSZ )
                  printf( "constant too long: truncated\n");
        else ungetc( c, stdin );
                /* above pushes back last char read */
        yylval.dval = atof ( buf );
        return( CONST );
        }
    return( c );
    }
```

```
INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
    /* returns the smallest interval containing a, b, c, and d */
    /* used by *, / routines */
    INTERVAL v;

    if( a>b ) { v.hi = a; v.lo = b; }
    else { v.hi = b; v.lo = a; }

    if( c>d ) {
        if ( c>v.hi ) v.hi = c;
        if ( d<v.lo ) v.lo = d;
        }
    else {
        if ( d>v.hi ) v.hi = d;
        if ( c<v.lo ) v.lo = c;
        }
    return( v );
    }

INTERVAL vmul( a, b, v ) double a, b; INTERVAL v; {
    return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
    }

dcheck( v ) INTERVAL v; {
    if( v.hi >= 0. && v.lo <= 0. ){
        printf( "divisor interval contains 0.\n" );
        return(1);
        }
    return(0);
    }

INTERVAL vdiv( a, b, v ) double a, b; INTERVAL v; {
    return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
    }
```

6.21 Old Features

This section mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotation marks (").

2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or underscores, the type number of the literal is defined, just as if the literal did not have the quotation marks around it. Otherwise, it is difficult to find the value for such

literals. The use of multicharacter literals is likely to mislead those unfamiliar with **yacc**, since it suggests that **yacc** is doing a job that must be actually done by the lexical analyzer.

3. Most places where '%' is legal, backslash (\) may be used. In particular, the double backslash (\\) is the same as *%%*, *\left* the same as *%left*, etc.

4. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec

5. Actions may also have the form

   ```
   ={...}
   ```

   and the curly braces can be dropped if the action is a single C statement.

6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Chapter 7

Using Signals

7.1 Introduction 7-1

7.2 Using the signal Function 7-1
7.2.1 Disabling a Signal 7-2
7.2.2 Restoring a Signal's Default Action 7-3
7.2.3 Catching a Signal 7-4
7.2.4 Restoring a Signal 7-6
7.2.5 Program Example 7-6

7.3 Catching Several Signals 7-7

7.4 Controlling Execution With Signals 7-8
7.4.1 Delaying a Signal's Action 7-8
7.4.2 Using Delayed Signals With System Functions 7-10
7.4.3 Using Signals in Interactive Programs 7-10

7.5 Using Signals in Multiple Processes 7-12
7.5.1 Protecting Background Processes 7-12
7.5.2 Protecting Parent Processes 7-13

7.1 Introduction

This chapter explains how to use C library functions to process signals sent to a program by the XENIX system. A signal is the system's response to an unusual condition that occurs during execution of a program such as a user pressing the INTERRUPT key or the system detecting an illegal operation. A signal interrupts normal execution of the program and initiates an action such as terminating the program or displaying an error message.

The **signal**(S) function of the standard C library lets a program define the action of a signal. The function can be used to disable a signal to prevent it from affecting the program. It can also be used to give a signal a user-defined action.

The **signal** function is often used with the **setjmp**(S) and **longjmp** (see **setime**(S)) functions to redefine and reshape the action of a signal. These functions allow programs to save and restore the execution state of a program, giving a program a means to jump from one state of execution to another without a complex assembly language interface.

To use the **signal** function, you must add the line

```
#include <signal.h>
```

to the beginning of the program. The *signal.h* file defines the various manifest constants used as arguments by the function. To use the **setjmp** and **longjmp** functions you must add the line

```
#include <setjmp.h>
```

to the beginning of the program. The *setjmp.h* file contains the declaration for the type **jmp_buf**, a template for saving a program's current execution state.

7.2 Using the signal Function

The *signal*() function changes the action of a signal from its current action to a given action. The function has the form

signal (*sigtype*, *ptr*);

where *sigtype* is an integer or a manifest constant that defines the signal to be changed, and *ptr* is a pointer to the function defining the new action or a manifest constant giving a predefined action. The function always returns a pointer value. This pointer defines the signal's previous action and may be used in subsequent calls to restore the signal to its previous value.

The *ptr* may be "SIG_IGN" to indicate no action (ignore the signal) or "SIG_DFL" to indicate the default action. The *sigtype* may be "SIGINT" for interrupt signal, caused by pressing the INTERRUPT key, "SIGQUIT" for quit signal, caused by pressing the QUIT key, or "SIGHUP" for hangup signal, caused by hanging up the line when connected to the system by modem. (Other constants for other signals are given in **signal**(S) in the XENIX *Reference Manual.*)

For example, the function call

```
signal(SIGINT, SIG_IGN);
```

changes the action of the interrupt signal to no action. The signal will have no effect on the program. The default action is usually to terminate the program.

The following sections show how to use the **signal** function to disable, change, and restore signals.

7.2.1 Disabling a Signal

You can disable a signal, i.e., prevent it from affecting a program, by using the "SIG_IGN" constant with **signal.** The function call has the form

signal (*sigtype*, SIG_IGN);

where *sigtype* is the manifest constant of the signal you wish to disable. For example, the function call

```
signal(SIGINT, SIG_IGN);
```

disables the interrupt signal.

The function call is typically used to prevent a signal from terminating a program executing in the background (e.g., a child process that is not using the terminal for input or output). The system passes signals generated from keystrokes at a terminal to all programs that have been invoked from that terminal. This means that pressing the INTERRUPT key to stop a program running in the foreground will also stop a program running in the background if it has not disabled that signal. For example, in the following program fragment, **signal** is used to disable the interrupt signal for the child.

```
#include <signal.h>

main ()
{

        if ( fork() == 0) {
                signal(SIGINT, SIG_IGN);
                /* Child process. */

        }

/* Parent process. */

}
```

This call does not affect the parent process which continues to receive interrupts as before. Note that if the parent process is interrupted, the child process continues to execute until it reaches its normal end.

7.2.2 Restoring a Signal's Default Action

You can restore a signal to its default action by using the "SIG_DFL" constant with **signal**. The function call has the form

```
signal (sigtype, SIGDFL);
```

where *sigtype* is the manifest constant defining the signal you wish to restore. For example, the function call

```
signal (SIGINT, SIG_DFL);
```

restores the interrupt signal to its default action.

The function call is typically used to restore a signal after it has been temporarily disabled to keep it from interrupting critical operations. For example, in the following program fragment the second call to **signal** restores the signal to its default action.

```
#include <signal.h>
#include <stdio.h>

main ()
{
        FILE *fp;
        char record[BUFSIZE], filename[100];

        signal (SIGINT, SIG_IGN);
        fp = fopen(filename, "a");
        fwrite(record, BUF, 1, fp);
        signal (SIGINT, SIG_DFL);

}
```

In this example, the interrupt signal is ignored while a record is written to the file given by "fp".

7.2.3 Catching a Signal

You can catch a signal and define your own action for it by providing a function that defines the new action and giving the function as an argument to **signal**. The function call has the form

signal (*sigtype*, *newptr*);

where *sigtype* is the manifest constant defining the signal to be caught, and *newptr* is a pointer to the function defining the new action. For example, the function call

```
signal(SIGINT, catch);
```

changes the action of the interrupt signal to the action defined by the function named *catch*().

The function call is typically used to let a program do additional processing before terminating. In the following program fragment, the function *catch*() defines the new action for the interrupt signal.

```
#include <signal.h>

main ()
{
        int catch ();

        printf("Press INTERRUPT key to stop.\n");
        signal (SIGINT, catch);
        while () {
                /* Body */
        }
}

catch ()
{
        printf("Program terminated.\n");
        exit(1);
}
```

The *catch*() function prints the message "Program terminated" before stopping the program with the **exit**(S) function.

A program may redefine the action of a signal at any time. Thus, many programs define different actions for different conditions. For example, in the following program fragment the action of the interrupt signal depends on the return value of a function named *keytest*.

```
#include <signal.h>

main ()
{
        int catch1 (), catch2 ();

        if (keytest() == 1)
                signal(SIGINT, catch1);
        else
                signal(SIGINT, catch2);

}
```

Later the program may change the signal to the other action or even a third action.

When using a function pointer in the **signal** call, you must make sure that the function name is defined before the call. In the program fragment shown above, *catch1* and *catch2* are explicitly declared at the beginning of the main program function. Their formal definitions are assumed to appear after the **signal** call.

7.2.4 Restoring a Signal

You can restore a signal to its previous value by saving the return value of a **signal** call, then using this value in a subsequent call. The function call has the form:

signal (*sigtype*, *oldptr*);

where *sigtype* is the manifest constant defining the signal to be restored and *oldptr* is the pointer value returned by a previous **signal** call.

The function call is typically used to restore a signal when its previous action may be one of many possible actions. For example, in the following program fragment the previous action depends solely on the return value of a function *keytest*.

```
#include <signal.h>

main ()
{
        int catch1(), catch2();
        int (*savesig)();

        if (keytest() == 1)
                signal(SIGINT, catch1);
        else
                signal(SIGINT, catch2);

        savesig = signal (SIGINT, SIG_IGN);
        compute();
        signal(SIGINT, savesig);

}
```

In this example, the old pointer is saved in the variable "savesig". This value is restored after the function *compute* returns.

7.2.5 Program Example

This section shows how to use the **signal** function to create a modifed version of the **system**(S) function. In this version, **system** disables all interrupts in the parent process until the child process has completed its operation. It then restores the signals to their previous actions.

```
#include <stdio.h>
#include <signal.h>

system(s)      /* run command strings */
char *s;
{
        int status, pid, w;
        register int (*istat)(), (*qstat)();

        if ((pid = fork()) == 0) {
                execl("/bin/sh", "sh", "-c", s, NULL);
                exit(127);
        }
        istat = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        while ((w = wait(&status)) != pid && w != -1)
                ;
        if (w == -1)
                status = -1;
        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        return(status);
}
```

Note that the parent uses the **while** statement to wait until the child's process ID "pid" is returned by **wait**(S). If **wait** returns the error code "−1" no more child processes are left, so the parent returns the error code as its own status.

7.3 Catching Several Signals

There are many more signals besides SIGINT, SIGQUIT, and SIGHUP. See the **signal**(S) manual page for a complete list. In the following program fragment, all signals are caught by the same function. This function makes use of the specific signal number which is passed as a parameter by the system.

```
#include <signal.h>

main()
{
        int i;
        int catch();

        for (i = 1; i <= NSIG; ++i)
                signal(i, catch);
        /*
         * Body
         */
}

catch(sig)
int sig;
{
        signal(sig, SIG_IGN);
        if (sig != SIGINT && sig != SIGQUIT && sig != SIGHUP)
                printf("Oh, oh.  Signal %d was received.\n", sig);
        unlink(tmpfile);
        exit(1);
}
```

The constant NSIG, the total number of signals, is defined in the file signal.h.

Note that the first action of the above catch function is to ignore the specific signal that was caught. This is necessary because the system automatically resets a caught signal to its default action.

7.4 Controlling Execution With Signals

Signals do not need to be used solely as a means of immediately terminating a program. Many signals can be redefined to delay their actions or even cause actions that terminate a portion of a program without terminating the entire program. The following sections describe ways that signals can be caught and used to provide control of a program.

7.4.1 Delaying a Signal's Action

You can delay the action of a signal by catching the signal and redefining its action to be nothing more than setting a globally-defined flag. Such a signal does nothing to the current execution of the program. Instead, the program continues uninterrupted until it can test the flag to see if a signal has been received. It can then respond according to the value of the flag.

The key to a delayed signal is that functions used to catch signals return execution to the exact point at which the program was interrupted. If the function returns normally the program continues execution just as if no signal occurred.

Delaying a signal is especially useful in programs that must not be stopped at an arbitrary point. If, for example, a program updates a linked list, the action of a signal can be delayed to prevent the signal from interrupting the update and destroying the list. For example, in the following program fragment the function *delay*() used to catch the interrupt signal sets the globally-defined flag "sigflag" and returns immediately to the point of interruption.

```
#include <signal.h>

int sigflag;

main ()
{
        int delay ();
        int (*savesig)();

        signal(SIGINT, delay); /* Delay the signal. */
        updatelist();
        savesig = signal(SIGINT, SIG_IGN); /* Disable the signal. */
        if (sigflag)
                /* Process delayed signals if any. */

}

delay ()
{
        signal(SIGINT, delay);
        sigflag=1;
}
```

In this example, if the signal is received while *updatelist* is executing, it is delayed until after *updatelist* returns. Note that the interrupt signal is disabled before processing the delayed signal to prevent a change to "sigflag" when it is being tested.

The first action of the delay function was to recatch the interrupt signal. This is necessary because the system resets caught signals to their default action which is normally immediate termination.

7.4.2 Using Delayed Signals With System Functions

When a delayed signal is used to interrupt the execution of a XENIX system function, such as *read*() or *wait*(), the system forces the function to stop and return an error code. This action, unlike actions taken during execution of other functions, causes all processing performed by the system function to be discarded. A serious error can occur if a program interprets a system function error caused by delayed signals as a normal error. For example, if a program receives a signal when reading the terminal, all characters read before the interruption are lost, making it appear as though no characters were typed.

Whenever a program intends to use delayed signals during calls to system functions, the program should include a check of the function return values to ensure that an error was not caused by an interruption. In the following program fragment, the program checks the current value of the interrupt flag "intflag" to make sure that the value EOF returned by *getchar* actually indicates the end of the file.

```
if (getchar() == EOF)
        if (intflag)
                /* EOF caused by interrupt */
        else
                /* true end-of-file */
```

7.4.3 Using Signals in Interactive Programs

Signals can be used in interactive programs to control the execution of the program's various commands and operations. For example, a signal may be used in a text editor to interrupt the current operation (e.g., displaying a file) and return the program to a previous operation (e.g., waiting for a command).

To provide this control, the function that redefines the signal's action must be able to return execution of the program to a meaningful location, not just the point of interruption. The standard C library provides two functions to do this: **setjmp** and **longjmp**. The **setjmp** function saves a copy of a program's execution state. The **longjmp** function changes the current execution state to a previously saved state. The functions cause a program to continue execution at an old location with old register values and status as if no operations had been performed between the time the state was saved and the time it was restored.

The **setjmp** function has the form

```
setjmp (buffer);
```

where *buffer* is the variable to receive the execution state. It must be explicitly declared with type **jmp_buf** before it is used in the call. For example, in the following program fragment **setjmp** copies the execution of the program to the variable "oldstate" defined with type **jmp_buf**.

```
jmp_buf oldstate;

setjmp(oldstate);
```

Note that after a **setjmp** call, the *buffer* variable contains values for the program counter, the data and address registers, and the process status. These values must not be modified in any way.

The **longjmp** function has the form

longjmp (*buffer*);

where *buffer* is the variable containing the execution state. It must contain values previously saved with a **setjmp** function. The function copies the values in the *buffer* variable to the program counter, data and address registers, and the process status table. Execution continues as if it had just returned from the **setjmp** function which saved the previous execution state. For example, in the following program fragment **setjmp** saves the execution state of the program at the location just before the main processing loop and **longjmp** restores it on an interrupt signal.

```
#include <signal.h>
#include <setjmp.h>

jmp_buf sjbuf;

main()
{
        int onintr();

        setjmp(sjbuf);
        signal(SIGINT, onintr);

        /* main processing loop */
}

onintr ()
{
        printf("\nInterrupt\n");
        longjmp(sjbuf);
}
```

In this example, the action of the interrupt signal as defined by *onintr* is to print the message "Interrupt" and restore the old execution state. When an interrupt signal is received in the main processing loop, execution passes to *onintr* which prints the message, then passes execution back to the main program function, making it appear as though control is returning from the **setjmp** function.

7.5 Using Signals in Multiple Processes

The XENIX system passes all signals generated at a given terminal to all programs invoked at that terminal. This means that a program has potential access to a signal even if that program is executing in the background or as a child to some other program. The following sections explain how signals may be used in multiple processes.

7.5.1 Protecting Background Processes

Any program that has been invoked using the shell's background symbol (&) is executed as a background process. Such programs usually do not use the terminal for input or output, and complete their tasks silently. Since these programs do not need additional input, the shell automatically disables the signals before executing the program. This means signals generated at the terminal do not affect execution of the program. This is how the shell protects the program from signals intended for other programs invoked from the same terminal.

In some cases, a program that has been invoked as a background process may also attempt to catch its own signals. If it succeeds, the protection from interruption given to it by the shell is defeated, and signals intended for other programs will interrupt the program. To prevent this, any program which is intended to be executed as a background process, should test the current state of a signal before redefining its action. A program should redefine a signal only if the signal has not been disabled. For example, in the following program fragment the action of the interrupt signal is changed only if the signal is not currently being ignored.

```
#include <signal.h>

main()
{
        int catch();

        if (signal(SIGINT, SIG_IGN) != SIG_IGN)
                signal(SIGINT, catch);

        /* Program body. */
}
```

This step lets a program continue to ignore signals if it is already doing so, and change the signal if it is not.

7.5.2 Protecting Parent Processes

A program can create and wait for a child process that catches its own signals if and only if the program protects itself by disabling all signals before calling the **wait** function. By disabling the signals, the parent process prevents signals intended for the child processes from terminating its call to **wait**. This prevents serious errors that may result if the parent process continues execution before the child processes are finished.

For example, in the following program fragment the interrupt signal is disabled in the parent process immediately after the child is created.

```
#include <signal.h>

main ()
{
        int (*saveintr)();

        if (fork () == 0)
                execl( ... );

        saveintr = signal (SIGINT, SIG_IGN);
        wait( &status );
        signal (SIGINT, saveintr);
}
```

The signal's action is restored after the **wait** function returns normal control to the parent.

Appendix A
M4: A Macro Processor

A.1 Introduction A-1

A.2 Invoking m4 A-1

A.3 Defining Macros A-2

A.4 Quoting A-3

A.5 Using Arguments A-5

A.6 Using Arithmetic Built-ins A-6

A.7 Manipulating Files A-7

A.8 Using System Commands A-8

A.9 Using Conditionals A-8

A.10 Manipulating Strings A-9

A.11 Printing A-11

A.1 Introduction

The **m4**(CP) macro processor defines and processes specially defined strings of characters called macros. By defining a set of macros to be processed by **m4**, a programming language can be enhanced to make it:

- More structured
- More readable
- More appropriate for a particular application

The **#define** statement in C and the analogous **define** in Ratfor are examples of the basic facility provided by any macro processor–replacement of text by other text.

Besides the straightforward replacement of one string of text by another, **m4** provides:

- Macros with arguments
- Conditional macro expansions
- Arithmetic expressions
- File manipulation facilities
- String processing functions

The basic operation of **m4** is copying its input to its output. As the input is read, each alphanumeric token (that is, string of letters and digits) is checked. If the token is the name of a macro, then the name of the macro is replaced by its defining text. The resulting string is reread by **m4**. Macros may also be called with arguments, in which case the arguments are collected and substituted in the right places in the defining text before **m4** rescans the text.

m4 provides a collection of about twenty built-in macros. In addition, the user can define new macros. Built-ins and user-defined macros work in exactly the same way, except that some of the built-in macros have side effects on the state of the process.

A.2 Invoking m4

The invocation syntax for **m4** is:

```
m4 [files]
```

Each file name argument is processed in order. If there are no arguments, or if an argument is a dash (-), then the standard is read. The processed text is written to the standard output, and can be redirected as in the following example:

```
m4 file1 file2 - > outputfile
```

Note the use of the dash in the above example to indicate processing of the standard input, *after* the files and have been processed by **m4**.

A.3 Defining Macros

The primary built-in function of **m4** is **define**, which is used to define new macros. The input:

define(*name*, *stuff*)

causes the string *name* to be defined as *stuff*. All subsequent occurrences of *name* will be replaced by *stuff*. *name* must be alphanumeric and must begin with a letter (the underscore (_) counts as a letter). *stuff* is any text, including text that contains balanced parentheses; it may stretch over multiple lines.

Thus, as a typical example:

```
define(N, 100)
.
.
.
if (i > N)
```

defines "N" to be 100, and uses this symbolic constant in a later **if** statement.

The left parenthesis must immediately follow the word **define**, to signal that **define** has arguments. If a macro or built-in name is not followed immediately by a left parenthesis, "(", it is assumed to have no arguments. This is the situation for "N" above; it is actually a macro with no arguments. Thus, when it is used, no parentheses are needed following its name.

You should also notice that a macro name is only recognized as such if it appears surrounded by nonalphanumerics. For example, in:

```
define(N, 100)
...
if (NNN > 100)
```

the variable "NNN" is absolutely unrelated to the defined macro "N", even though it contains three N's.

Things may be defined in terms of other things. For example:

```
define(N, 100)
define(M, N)
```

defines both M and N to be 100.

What happens if "N" is redefined? Or, to say it another way, is: "M" defined as "N" or as 100? In **m4**, the latter is true, "M" is 100, so even if "N" subsequently changes, "M" does not.

This behavior arises because **m4** expands macro names into their defining text as soon as it possibly can. Here, that means that when the string "N" is seen as the arguments of **define** are being collected, it is immediately replaced by 100; it is just as if you had said:

```
define(M, 100)
```

in the first place.

If this isn't what you really want, there are two ways out of it. The first, which is specific to this situation, is to interchange the order of the definitions:

```
define(M, N)
define(N, 100)
```

Now "M" is defined to be the string "N", so when you ask for "M" later, you will always get the value of "N" at that time (because the "M" will be replaced by "N" which, in turn, will be replaced by 100).

A.4 Quoting

The more general solution is to delay the expansion of the arguments of **define** by quoting them. Any text surrounded by single quotation marks ` and ´ is not expanded immediately, but has the quotation marks stripped off. If you say:

```
define(N, 100)
define(M, 'N')
```

the quotation marks around the "N" are stripped off as the argument is being collected, but they have served their purpose, and "M" is defined as the string "N", not 100. The general rule is that **m4** always strips off one level of single quotation marks whenever it evaluates something. This is true even outside of macros. If you want the word "define" to appear in the output, you have to quote it in the input, as in:

```
'define' = 1;
```

As another instance of the same thing, which is a bit more surprising, consider redefining "N":

```
define(N, 100)
...
define(N, 200)
```

Perhaps regrettably, the "N" in the second definition is evaluated as soon as it is seen; that is, it is replaced by 100, so it's as if you had written:

```
define(100, 200)
```

This statement is ignored by **m4**, since you can only define things that look like names, but it obviously does not have the effect you wanted. To really redefine "N", you must delay the evaluation by quoting:

```
define(N, 100)
...
define('N', 200)
```

In **m4**, it is often wise to quote the first argument of a macro.

If the forward and backward quotation marks (` and ´) are not convenient for some reason, the quotation marks can be changed with the built-in **changequote**. For example:

```
changequote([, ])
```

makes the new quotation marks the left and right brackets. You can restore the original characters with just:

```
changequote
```

There are two additional built-ins related to **define**. The built-in **undefine** removes the definition of some macro or built-in:

```
undefine('N')
```

removes the definition of "N". Built-ins can be removed with **undefine**, as in:

```
undefine('define')
```

but once you remove one, you can never get it back.

The built-in **ifdef** provides a way to determine if a macro is currently defined. For instance, pretend that either the word "xenix" or "unix" is defined according to a particular implementation of a program. To perform operations according to which system you have you might say:

```
ifdef('xenix', 'define(system,1)')
ifdef('unix', 'define(system,2)')
```

Do not forget the quotation marks in the above example.

Ifdef actually permits three arguments: if the name is undefined, the value of **ifdef** is then the third argument, as in:

```
ifdef('xenix', on XENIX, not on XENIX)
```

A.5 Using Arguments

So far we have discussed the simplest form of macro processing – replacing one string by another (fixed) string. User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**) any occurrence of *$n* will be replaced by the *n*th argument when the macro is actually used. Thus, the macro *bump*, defined as:

```
define(bump, $1=$1+1)
```

generates code to increment its argument by 1:

```
bump(x)
```

is:

```
x=x+1
```

A macro can have as many arguments as you want, but only the first nine are accessible, through $1 to $9. (The macro name itself is $0.) Arguments that are not supplied are replaced by null strings, so we can define a macro *cat* which simply concatenates its arguments, like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus:

```
cat(x, y, z)
```

is equivalent to:

```
xyz
```

The arguments $4 through $9 are null, since no corresponding arguments were provided.

Leading unquoted blanks, tabs, or newlines that occur during argument collection are discarded. All other white space is retained. Thus:

```
define(a, b  c)
```

defines "a" to be "b   c".

Arguments are separated by commas, but parentheses are counted properly, so a comma protected by parentheses does not terminate an argument. That is, in:

```
define(a, (b,c))
```

there are only two arguments; the second is literally "(b,c)". And of course a bare comma or parenthesis can be inserted by quoting it.

A.6 Using Arithmetic Built-ins

m4 provides two built-in functions for doing arithmetic on integers. The simplest is **incr**, which increments its numeric argument by 1. Thus, to handle the common programming situation where you want a variable to be defined as one more than N, write:

```
define(N, 100)
define(N1, 'incr(N)')
```

Then "N1" is defined as one more than the current value of "N".

The more general mechanism for arithmetic is a built-in called **eval**, which is capable of arbitrary arithmetic on integers. It provides the following operators (in decreasing order of precedence):

```
unary + and -
** or ^ (exponentiation)
* / % (modulus)
+ -
== != < <= > >=
!  (not)
& or &&        (logical and)
| or ||  (logical or)
```

Parentheses may be used to group operations where needed. All the operands of an expression given to **eval** must ultimately be numeric. The numeric value of a true relation (like 1>0) is 1, and false is 0. The precision in **eval** is implementation dependent.

As a simple example, suppose we want "M" to be "2**N+1". Then:

```
define(N, 3)
define(M, 'eval(2**N+1)')
```

As a matter of principle, it is advisable to quote the defining text for a macro unless it is very simple indeed (say just a number); it usually gives the result you want, and is a good habit to get into.

A.7 Manipulating Files

You can include a new file in the input at any time by the built-in function **include**:

include(*filename*)

inserts the contents of *filename* in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (that is, its replacement text) is the contents of the file; this can be captured in definitions, etc.

It is a fatal error if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used; **sinclude** (for "silent include") says nothing and continues if it cannot access the file.

It is also possible to divert the output of **m4** to temporary files during processing, and output the collected material upon command. **m4** maintains nine of these diversions, numbered 1 through 9.

If you say:

```
divert(n)
```

all subsequent output is put onto the end of a temporary file referred to as "n". Diverting to this file is stopped by another **divert** command; in particular, **divert** or **divert(0)** resumes the normal output process.

Diverted text is normally output all at once at the end of processing, with the diversions output in numeric order. It is possible, however, to bring back diversions at any time, that is, to append them to the current diversion.

```
undivert
```

brings back all diversions in numeric order, and **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted stuff, as does diverting into a diversion whose number is not between 0 and 9 inclusive.

The value of **undivert** is not the diverted stuff. Furthermore, the diverted material is not rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. This is zero during normal processing.

A.8 Using System Commands

You can run any program in the local operating system with the **syscmd** built-in. For example,

```
syscmd(date)
```

runs the **date** command. Normally, **syscmd** would be used to create a file for a subsequent **include**.

To facilitate making unique file names, the built-in **maketemp** is provided, with specifications identical to the system function **mktemp**: a string of "XXXXX" in the argument is replaced by the process id of the current process.

A.9 Using Conditionals

There is a built-in called **ifelse** which enables you to perform arbitrary conditional testing. In the simplest form,

ifelse(*a*, *b*, *c*, *d*)

compares the two strings *a* and *b*. If these are identical, **ifelse** returns the string *c*; otherwise it returns *d*. Thus, we might define a macro called **compare** which compares two strings and returns "yes" or "no" if they are the same or different.

```
define(compare, 'ifelse($1, $2, yes, no)')
```

Note the quotation marks, which prevent too-early evaluation of **ifelse**.

If the fourth argument is missing, it is treated as empty.

ifelse can actually have any number of arguments, and thus provides a limited form of multi-way decision capability. In the input:

ifelse(*a*, *b*, *c*, *d*, *e*, *f*, *g*)

if the string *a* matches the string *b*, the result is *c*. Otherwise, if *d* is the same as *e*, the result is *f*. Otherwise the result is *g*. If the final argument is omitted, the result is null, so:

ifelse(*a*, *b*, *c*)

is *c* if *a* matches *b*, and null otherwise.

A.10 Manipulating Strings

The built-in **len** returns the length of the string that makes up its argument. Thus:

```
len(abcdef)
```

is 6, and:

```
len((a,b))
```

is 5.

The built-in **substr** can be used to produce substrings of strings. For example:

substr(*s*,*i*,*n*)

returns the substring of *s* that starts at position *i* (origin zero), and is *n* characters long. If *n* is omitted, the rest of the string is returned, so:

```
substr('now is the time', 1)
```

is:

```
ow is the time
```

If *i* or *n* are out of range, various sensible things happen.

The command:

```
index(s1,s2)
```

returns the index (position) in *s1* where the string *s2* occurs, or −1 if it does not occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration.

```
translit(s, f, t)
```

modifies *s* by replacing any character found in *f* by the corresponding character of *t*. That is:

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If *t* is shorter than *f*, characters that do not have an entry in *t* are deleted; as a limiting case, if *t* is not present at all, characters from *f* are deleted from *s*. So:

```
translit(s, aeiou)
```

deletes vowels from "s".

There is also a built-in called **dnl** which deletes all characters that follow it up to and including the next newline. It is useful mainly for throwing away empty lines that otherwise tend to clutter up **m4** output. For example, if you say:

```
define(N, 100)
define(M, 200)
define(L, 300)
```

the newline at the end of each line is not part of the definition, so it is copied into the output, where it may not be wanted. If you add **dnl** to each of these lines, the newlines will disappear.

Another way to achieve this, is:

```
divert(-1)
        define(...)
        ...
divert
```

A.11 Printing

The built-in **errprint** writes its arguments out on the standard error file. Thus, you can say:

```
errprint(‘fatal error’)
```

Dumpdef is a debugging aid that dumps the current definitions of defined terms. If there are no arguments, you get everything; otherwise you get the ones you name as arguments. Do not forget the quotation marks.

Appendix B
XENIX System Calls

B.1 Introduction B-1

B.2 Executable File Format B-1

B.3 Revised System Calls B-1

B.4 Version 7 Additions B-3

B.5 Changes to the ioctl Function B-3

B.6 Pathname Resolution B-3

B.7 Using the mount () and chown () Functions B-4

B.8 Super-Block Format B-4

B.9 Separate Version Libraries B-4

B.1 Introduction

This appendix lists some of the differences between XENIX 2.3, XENIX 3.0, UNIX V7, UNIX System 3.0 and XENIX System V. It is intended to aid users who wish to convert system calls in existing application programs for use on other systems.

B.2 Executable File Format

XENIX 3.0, UNIX System 3.0, and XENIX System V execute only those programs with the *x.out* executable file format. The format is similar to the old *a.out* format, but contains additional information about the executable file such as text and data relocation bases, target machine identification, word and byte ordering, symbol table, and relocation table format. The *x.out* file also contains the revision number of the kernel which is used during execution to control access to system functions. XENIX System V has a segmented *x.out* header which contains segmentation information, as well as relocation information. To execute existing programs in *a.out* format, you must first convert to the *x.out* format. The format is described in detail in **a.out**(F) in the XENIX *Reference Manual.*

XENIX System V uses little-endian (low order word first in memory) word order for longs whereas some XENIX 3.0 systems use big-endian (high order word first in memory) word order. XENIX System V checks the *x.out* header for information about the word order. XENIX System V maintains full XENIX 3.0 binary compatibility. XENIX System V executes XENIX 3.0 word-swapped (big-endian) executable files as well as XENIX 3.0 and XENIX System V (little-endian) executables. Refer to the **machine**(M) manual page in the XENIX *Reference Manual* for a complete description of binary compatibility.

B.3 Revised System Calls

Some system calls in XENIX System V and UNIX System V have been revised and do not perform the same tasks as the corresponding calls in previous systems. To provide compatibility for old programs, XENIX System V and UNIX System V maintain both the new and the old system calls and automatically check the revision information in the *x.out* header to determine which version of a system call should be made. The following table lists the revised system calls and their previous versions.

System Call #	XENIX 2.3 function	System 3 function	System V function
35	ftime	unused	unused
38	unused	clocal	clocal
39	unused	setpgrp	setpgrp
40	unused	cxenix	cxenix
57	unused	utssys	utssys
62	clocal	fcntl	fcntl
63	cxenix	ulimit	ulimit

The *cxenix*() function provides access to system calls unique to XENIX 3.0 and/or XENIX System V. The *clocal* funtion provides access to all calls unique to an OEM.

The new XENIX System V system calls are accessed via *cxenix*() system calls with their numbers. Note that these numbers are not regular system call numbers, but *cxenix*() numbers. To use these calls, the *cxenix*() system call is made, with the high byte set to the appropriate number listed below (i.e., to call *locking*, take 40, add 256*1 to it, and pass the resulting value in **ax** when trapping into the kernel.) The XENIX 3.0 and System V system calls are listed at the below.

These calls are valid for XENIX 3.0 and XENIX System V:

cxenix Call #	Function	System Call
0	shutdown OS	shutdown
1	record locking	locking
2	create semaphore	creatsem
3	open semaphore	opensem
4	signal semaphore	sigsem
5	wait semaphore	waitsem
6	nonblocking waitsem	nbwaitsem
7	blocking read check	rdchk
8	set stack limit	stkgrow
9	extended ptrace	xptrace
10	change file size	chsize
11	XENIX 2.3 ftime call	ftime
12	sleep for short interval	nap
13	attach to shared data	sdget
14	release shared data	sdfree
15	enter critical region	sdenter
16	leave critical region	sdleave
17	get shared data version #	sdgetv
18	wait for new shared data version	sdwaitv

The following calls are found in XENIX System V only:

cxenix Call #	Function	System Call
19	change segment size	brkctl
22	message control	msgctl
23	get message queue	msgget
24	send message	msgsnd
25	receive message	msgrcv
26	semaphore control	semctl
27	get semaphore set	semget
28	semaphore ops	semop
29	sysV shared memory control	shmctl
30	sysV create shared memory	shmget
31	sysV attach shared memory	shmat

B.4 Version 7 Additions

XENIX System V maintains a number of XENIX 3.0 and UNIX V7 features that were dropped from UNIX System 3.0. In particular, XENIX System V continues to support the **dup2**(S) and **ftime**(S) functions. The **ftime** function, used with the **ctime**(S) function, provides the default value for the time zone when the TZ environment variable has not been set. This means a binary configuration program can be used to change the default time zone. No source license is required.

B.5 Changes to the ioctl Function

XENIX 3.0 and UNIX System 3.0 have a full set of XENIX 2.3-compatible **ioctl** calls. Furthermore, XENIX 3.0 and XENIX System V have resolved problems that previously hindered UNIX System 3.0 compatibility. For convenience, XENIX 2.3-compatible **ioctl** calls can be executed by a UNIX System 3.0 executable. The available XENIX 2.3 **ioctl** calls are: TIOCSETP, TIOCSETN, TIOCGETP, TIOCSETC, TIOCGETC, TIOCEXCL, TIOCNXCL, TIOCHPCL, TIOCFLUSH, TIOCGETD, and TIOCSETD.

B.6 Pathname Resolution

If a null pathname is given, XENIX 2.3 interprets the name to be the current directory, but UNIX System 3.0 considers the name to be an error. XENIX 3.0 and XENIX System V use the version number in the *x.out* header to determine what action to take. A XENIX 2.3 header causes null pathnames to be the current directory. Any other version is interpreted as an error.

If the symbol ".." is given as a pathname when in a root directory that has been defined using the **chroot**(S) function, XENIX 2.3 moves to the next higher directory. XENIX 3.0 also allows the ".." symbol to **chroot**, but restricts its use to the super-user. XENIX System V does not allow the ".." symbol to **chroot**.

B.7 Using the mount () and chown () Functions

XENIX 3.0, and UNIX System 3.0 restrict the use of the **mount**(S) system call to the super-user. XENIX System V does not restrict the use of the **mount** system call, usually however, the **mount**(C) program is only executable by the super-user. Also, XENIX System V, 3.0 and UNIX System 3.0 allow the owner of a file to use **chown**(S) function to change the file ownership.

B.8 Super-Block Format

XENIX System V, UNIX System 3.0 and UNIX System 5.0 have new super-block formats. XENIX System V and XENIX 3.0 use the System 5.0 format, but use a different magic number for each revision. The XENIX System V and XENIX 3.0 super-blocks have an additional field at the end which can be used to distinguish between XENIX 2.3, 3.0 and System V super-blocks. XENIX System V and XENIX 3.0 check this magic number at boot time and during a mount. If a XENIX 2.3 super-block is read, XENIX 3.0 converts it to the new format internally. Similarly, if a XENIX 2.3 super-block is written, XENIX 3.0 converts it back to the old format. This permits XENIX 2.3 kernels to be run on file systems also usable by UNIX System 3.0.

However, XENIX System V is word-swapped relative to XENIX-86 3.0. Even though the super-block formats are the same, the order of bytes in long words is different. XENIX System V can not **mount**(C) or **fsck**(C) XENIX 3.0 filesystems.

B.9 Separate Version Libraries

XENIX System V supports the construction of XENIX 3.0 executable files. This systems maintains both the new and old versions of system calls in separate libraries.

Index

A

ar
  description 1-2
As
  basic tool 1-1
Assembler See As

C

C compiler
  expression
    evaluation order 4-12
  lint directives, effect 4-13
C language
  yacc 6-1
C program
  string extraction 1-3
C programming language 1-1
C-shell
  command history mechanism 1-3
  command language 1-3
Command
  execution 1-3
  interpretation 1-3
  SCCS commands See SCCS
  SCCS See SCCS
csh
  description 1-3

D

Delta See SCCS
Desk calculator
  specifications 6-32
Development system
  described 1-1

E

Error message file
  creation 1-2

F

File
  archives 1-2
  block counting 1-2
  check sum computation 1-2
  error message file See Error message file
  octal dump 1-2
  relocation bits removal 1-2
  removal
  Source Code Control System See SCCS
  symbol removal 1-2
  text search, print 1-2
FORTRAN
  conversion program 5-21

H

Hexadecimal dump 1-2

L

ld
  basic tool 1-1
lex
  -ll flag
    library access 5-4
  0, end of file notation 5-12
  a.out file
    contents 5-4
  action
    default 5-8
    repetition 5-8
    specification 5-8
  alternation 5-6
  ambiguous source rules 5-12
  angle brackets (<)
    operator character 5-3, 5-25
    start condition referencing 5-16
  arbitrary character match 5-5
  array size change 5-24
  asterisk (*)
    operator character 5-3, 5-25
    repeated expression specification 5-6
  automaton interpreter
    initial condition resetting 5-16
  backslash (\)
    C escapes 5-3

lex *(continued)*
  backslash (\) *(continued)*
    operator character 5-3, 5-25
    operator character escape 5-3, 5-3, 5-5

  BEGIN
    start condition entry 5-16
  blank character
    quoting 5-3
    rule ending 5-3
  blank, tab line beginning 5-17
  braces ({})
    expression repetition 5-7
    operator character 5-3, 5-25
  brackets ([])
    character class specification 5-4
    character class use 5-1
    operator character 5-3, 5-25
    operator character escape 5-4
  buffer overflow 5-13
  C escapes 5-3
  caret (^) operator
    left context recognizing 5-15
    character class inclusion 5-5
    context sensitivity 5-6
    operator character 5-3, 5-25
    string complement 5-5
  character class
    notation 5-1
    specification 5-4
  character set
    specification 5-23
    internal use 5-23
    set table 5-23, 5-23, 5-24
    translation table See set table
  context sensitivity 5-6
  copy classes 5-17
  dash (-)
    character class inclusion 5-5
    operator character 5-3
    range indicator 5-4
  dash (–)
    operator character 5-25
  definition
    expansion 5-7
    format 5-18
    placement 5-7
  definitions
    character set table 5-23
    contents 5-18, 5-24
    format 5-24
    location 5-18
    specification 5-17
  delimiter
    discard 5-18
    rule beginning marking 5-1
    third delimiter, copy 5-18
  description 1-1, 5-1
  dollar sign ($) operator

lex *(continued)*
  dollar sign ($) operator *(continued)*
    right context recognizing 5-15
    context sensitivity 5-6
    end of line notation 5-1
    operator character 5-3, 5-25
  dot (.) operator See period (.)
  double precision constant change 5-21
  ECHO
    format argument, data printing 5-9
  end-of-file
    0 handling 5-12
    yywrap routine 5-12
  environment
    change 5-15
  expression
    new line illegal 5-4
    repetition 5-7
  external character array 5-8
  flag
    environment change 5-15
  FORTRAN conversion program 5-21
  grouping 5-6
  I/O library See library
  I/O routine
    access 5-11
    consistency 5-11
  input (\) routine 5-11
  input routine
    character I/O handling 5-23
    description 5-1
    end-of-file, 0 notation 5-11
    ignoring 5-8
    manipulation restriction 5-15
  invocation 5-4
  left context 5-7
    caret (^) operator 5-15
    sensitivity 5-15
  lex.yy.c file 5-4
  lexical analyzer
    environment change 5-15
  library
    access 5-4
    avoidance 5-4
    backup limitation 5-12
    loading 5-19
  line beginning match 5-6
  line end match 5-7
  loader flag See -ll flag
  lookahead characteristic 5-9, 5-11
  match count 5-9
  matching
    occurrence counting 5-13
    preferences 5-12
  new line
    illegality 5-4
  newline
    escape 5-23
    matching 5-13

lex *(continued)*
octal escape 5-5
operator character
escape 5-3
quoting 5-3
operator characters
designated 5-25
escape 5-4, 5-5
listing 5-3
literal meaning 5-3
optional expression
specification 5-5
output (c) routine 5-11
output routine
character I/O handling 5-23
parentheses (())
grouping 5-6
operator character 5-3
parenthesis (())
operator character 5-25
parser generator
analysis phase 5-2
percentage sign (%)
delimiter notation (%%) 5-1
operator character 5-3
remainder operator 5-19
source segment separator 5-8
period (.) operator
designted 5-25
arbitrary character match 5-5
newline no match 5-13
operator character 5-3
plus sign (+)
operator character 5-3, 5-25
repeated expression specification 5-6
preprocessor statement entry 5-18
question mark (?)
operator character 5-3, 5-25
optional expression specification 5-5
quotation marks, double (5-3, 5-25
real numbers rule 5-18
regular expression
description 5-3
operators See operator characters
REJECT 5-14, 5-14
repeated expression
specification 5-6
right context
dollar sign ($) operator 5-15
rule
active 5-16
real number 5-18
rules
format 5-25
semicolon (;)
null statement 5-8
slash (/)
operator character 5-3, 5-25
trailing text 5-7

lex *(continued)*
source definitions
specification 5-17
source file
format 5-24
source program
compilation 5-4
copy into generated program 5-17
description 5-1
format 5-17
interception failure 5-17
segment separator 5-8
spacing character ignoring 5-8
start condition 5-7
entry 5-16
environment change 5-15
start conditions
format 5-24
location 5-24
abbreviation 5-16
statistics gathering 5-20
substitution string
definition See definition
tab line beginning See blank, tab line beginning
text character
quoting 5-3
trailing text 5-7
unput (c) routine 5-11
unput routine
character I/O handling 5-23
REJECT noncompatible 5-15
unreachable statement 4-6
vertical bar (|)
action repetition 5-8
operator character 5-3, 5-25
vertical bar (|)
alternation 5-6
wrapup See yywrap routine
yacc interface
tokens 5-19
yylex (\) 5-19
interface 5-2
library loading 5-19
yyleng variable 5-9
yyless (n) 5-9
yyless (\)
text reprocessing 5-10
yylex (\) program
yacc interface 5-19
yylex program
contents 5-1
yymore (\) 5-9
yytext
external character array 5-8
yywrap () 5-20
yywrap (\) routine 5-12
Library
conversion 1-2

Library *(continued)*
  maintenance 1-2
  ordering relation 1-2
  sort 1-2
lint
  -a option 4-9
  -b option 4-5
  -c option 4-8
  -ly directive 4-14
  -n option 4-14
  -p option 4-14
  -u option 4-4
  -v option
  -x option 4-4
    turnon 4-13
    unused variable report suppression 4-4
  ARGSUSED directive 4-13, 4-14
  argument number comments turnoff 4-13

  assignment of long to int
    check 4-9
  assignment operator
    new form 4-11
    old form, check 4-11
    operand type balancing 4-7
  assignment, implied See implied assignment

  binary operator, type check 4-7
  break statement
    unreachable See unreachable break
        statement
  C program check 4-1
  C syntax, old form, check 4-11
  cast See type cast
  conditional operator, operand type balancing
      4-7
  constant in conditional context 4-10
  construction check 4-1, 4-9
  control information flow 4-13
  degenerate unsigned comparison 4-10
  description 4-1
  directive
    defined 4-13
    embedding 4-13
  enumeration, type check 4-7
  error message, function name 4-7
  expression, order 4-12
  extern statement 4-3
  external declaration, report suppression 4-4

  file
    library declaration file identification 4-13

  function
    error message 4-7
    return value check 4-6
    type check 4-7
    unused See unused function
  implied assignment, type check 4-7

lint *(continued)*
  initialization, old style check 4-11
  library
    compatibility check 4-14
    compatibility check suppression 4-14
    directive acceptance 4-14
    file processing 4-14
  LINTLIBRARY directive 4-13, 4-14
  loop check 4-6
  nonportable character check 4-9
  nonportable expression evaluation order
      check 4-12
  NOSTRICT directive 4-13
  NOTREACHED directive 4-13
  operator
    operand types balancing 4-7
    precedence 4-10
  output turnoff 4-13
  pointer
    agreement 4-7
    alignment check 4-12
  program flow
    control 4-5
  relational operator, operand type balancing
      4-7
  scalar variable check 4-12
  source file, library compatibility check 4-14

  statement, unlabeled report 4-5
  structure selection operator, type check 4-7

  syntax 4-1
  type cast
    check 4-8
    comment printing control 4-8
  type check
    description 4-7
    turnoff 4-13
  unreachable break statement, report
      suppression 4-6
  unused argument
    report suppression 4-4
  unused function, check 4-3
  unused variable, check 4-3
  VARARGS directive 4-13, 4-14
  variable
    external variable initialization 4-5
    inner/outer block conflict 4-10
    set/used information 4-4
    static variable initialization 4-5
    unused See unused variable
  –h option 4-11
Loader See ld
longjmp function 7-11
Loop
  lint use See lint
lorder
  description 1-2

M

m4
  description 1-2
Macros
  preprocessing 1-2
Maintainer See Make
make command
  arguments 2-4
  syntax 2-4
  -d option 2-14
  -e option 2-4
  -i option 2-4
  -k option 2-4
  -n option 2-4, 2-14
  -p option 2-4
  -r option 2-4
  -s option 2-4
  -t option 2-4, 2-14
  .c suffix 2-10
  .DEFAULT 2-5
  .f suffix 2-10
  .IGNORE 2-5
  .l suffix 2-10
  .o suffix 2-10
  .PRECIOUS 2-5
  .r suffix 2-10
  .s suffix 2-10
  .SILENT 2-5
  .y suffix 2-10
  .yr suffix 2-10
  argument quoting 2-7
  backslash (\)
    description file continuation 2-2
Make
  basic tool 1-2
make
  command argument
    macro definition 2-6
  command string substitution 2-5
    hyphen (-) start 2-5
    form 2-1
    location 2-1
    print without execution 2-14
  dependency line substitution 2-5
    form 2-1
  description file
    comment convention 2-2
    macro definition 2-6
  description filename
    argument 2-4
  dollar sign ($)
    macro invocation 2-6
  equal sign (=)
    macro definition 2-6

make *(continued)*
  file generation 2-5
  file update 2-1
    time, date printing 2-14
    updating 2-14
  hyphen (-)
    command string start 2-5
  macro definition
    analysis 2-6
    argument 2-4
    description 2-6
    definition 2-6
    definition override 2-7
    invocation 2-6
    substitution 2-5, 2-5
    value assignment 2-6
  medium sized projects 2-1
  metacharacter expansion 2-1
  number sign (#)
    description file comment 2-2
  object file
    suffix 2-10
  option argument
    use 2-4
  parentheses (())
    macro enclosure 2-6
  program maintenance 2-1
  semicolon (;)
    command introduction 2-1
  source file
    suffixes 2-10
  source grammar
    suffixes 2-10
  suffixes
    list 2-9
    table 2-9
  target file
    pseudo-target files 2-5
    update 2-14
  target filename
    argument 2-4
  target name omission 2-3
  touch option See -t option
  transformation rules
    table 2-9
  troubleshooting 2-13

N

Notational conventions 1-4

P

Pipe
  SCCS use See SCCS
Processes
  background 7-12
  restoring an execution state 7-11
  saving the execution state 7-10
Program development 1-1
  maintainer See Make

R

ranlib
  description 1-2
rm command
  SCCS use See SCCS

S

SCCS, source code control 1-2
  %M% keyword
    g-file line precedence 3-31
  -a option
    login name addition use 3-24
  -d flag
    flags deletion 3-17
  -d option
    data specification provision 3-21
    flag removal 3-17
  -e option
    delta range printing 3-22
    file editing use 3-7
    login name removal 3-25
  -f option
    flag initialization, modification 3-16
    flag, value setting 3-17
  -g option
    output suppression 3-32
    p-file regeneration 3-27
  -h option
    file audit use 3-26
  -i flag
    keyword message, error treatment 3-16
  -i option
    delta inclusion list use 3-30
  -k option
    g-file regeneration 3-27
  -l option
    delta range printing 3-22
    l-file creation 3-31
  -m option

SCCS, source code control 1-2 *(continued)*
  -m option *(continued)*
    effective when 3-19
    file change identification 3-31
    new file creation 3-29
  -n option
    %M% keyword value use 3-31
    g-file preservation 3-13
    pipeline use 3-31
  -p option
    delta printing 3-32
    output effect 3-12
  -r option
    delta creation use 3-23
    delta printing use 3-22
    file retrieval 3-9
    release number specification 3-29
  -s option
    output suppression 3-29, 3-29
  -t option
    delta retrieval 3-12
    file initialization 3-21
    file modification 3-21
  -x option
    delta exclusion list use 3-30
  -y option
    comments prompt response 3-18
    new file creation 3-29
  -z key
    file audit use 3-27
  @(#) string
    file information, search 3-33
  admin command
    file administration 3-26
    file checking use 3-26
    file creation 3-5
    use authorization 3-6
  administrator
    description 3-5
  argument
    minus sign(-)
      use 3-4
    types designated 3-4
  branch delta
    retrieval 3-11
  branch number
    description 3-2
  cdc command
    commentary change 3-18
  ceiling flag
    protection 3-25, 3-25
  checksum
    file corruption determination 3-26
  command
    argument See argument
    execution control 3-4
    explanation 3-28
  comments

SCCS, source code control 1-2 *(continued)*
 comments *(continued)*
  change procedure 3-18
  omission, effect 3-29
 corrupted file
  determination 3-26
  processing restrictions 3-26
  restoration 3-27
 d flag
  default specification 3-17
 d-file
  temporary g-file 3-4
 data keyword
  data specification component 3-21
  replacement 3-21
 data specification
  description 3-21
 delta command
  comments prompt 3-8
  file change procedure 3-8
  g-file removal 3-13
  p-file reading 3-8, 3-8
 delta table
  delta removal, effect 3-33
  description 3-18
  branch delta See branch delta
  defined 3-1, 3-2
  exclusion 3-30
  inclusion 3-30
  interference 3-30
  latest release retrieval 3-12
  level number See level number
  name See SID
  printing 3-22, 3-32
  range printing 3-22
  release number See release number
  removal 3-32
 descriptive text
  initialization 3-20
  modification 3-20
  removal 3-21
 diagnostic output
  -p option effect 3-13
 diagnostics
  code as help argument 3-13
  form 3-13
 directory use 3-2
  file argument application 3-4
  x-file location 3-3
 error message
  code use 3-13
  form 3-13
 exclamation point (!)
  MR deletion use 3-20
 file argument
  description 3-4
  processing 3-4
 file creation

SCCS, source code control 1-2 *(continued)*
 file creation *(continued)*
  comment line generation 3-29
  commentary 3-29
  comments omission, effect 3-29
  level number 3-29
  release number 3-29
 file protection 3-24
  administration 3-26
  change identification 3-31
  change procedure 3-8
  change, major 3-10
  changes See delta
  checking procedure 3-26
  comparison 3-33
  composition 3-2, 3-18
  corrupted file See corrupted file
  creation 3-5
  data keyword See data keyword
  descriptive text description 3-18
  descriptive text See descriptive text
  editing, -e option use 3-7
  grouping 3-2
  identifying information 3-33
  link See link
  multiple concurrent edits 3-23
  name arbitrary 3-13
  name See link
  name, s use 3-5
  parameter initialization, modification 3-

  printing 3-21
  protection methods 3-24
  removal 3-6
  retrieval See get command
  x-file See x-file
 flags
  deletion 3-17
  initialization 3-16
  modification 3-16
  setting, value setting 3-17
  use 3-17
 floor flag
  protection 3-25
 g-file
  creation 3-3
  creation date, time recordation 3-14
  description 3-3
  line identification 3-31
  line, %M% keyword value 3-31
  ownership 3-3
  regeneration 3-27
  removal, delta command use 3-13
  temporary See d-file
 get command
  -e option use 3-7
  concurrent editing, directory use 3-23
  delta inclusion, exclusion check 3-30

SCCS, source code control 1-2
(continued)
get command *(continued)*
file retrieval 3-6
filename creation 3-6
g-file creation 3-3
message 3-6
release number change 3-10
help command
argument 3-13
code use 3-13
use 3-28
i flag
file creation, effect 3-16
ID keyword See keyword
identification string See SID
j flag
multiple concurrent edits specification 3-23
keyword
data See data keyword
format 3-14
lack, error treatment 3-16
use 3-14
l-file
contents 3-4
creation 3-31
level number
delta component 3-2
new file 3-29
omission, file retrieval, effect 3-10
link
number restriction 3-2
lock file See z-file
lock flag
R protection 3-25
minus sign (-)
option argument use 3-4
minus sign(-)
argument use 3-4
mode
g-file 3-3
MR
commentary supply 3-18
deletion 3-20
new file creation 3-29
multiple users 3-5
option argument
description 3-4
processing order 3-4
output
data specification See data specification

piping 3-29
suppression, -g option 3-32
suppression, -s option 3-29, 3-29
write to standard output 3-12
p-file
contents 3-3, 3-8

SCCS, source code control 1-2
(continued)
p-file *(continued)*
creation 3-3
delta command reading 3-8
naming 3-3
ownership 3-3
permissions 3-3
regeneration 3-27
update 3-3
updating 3-4
percentage sign (%)
keyword enclosure 3-14
piping 3-29
-n option use 3-31
prs command
file printing 3-21
purpose 3-1
q file
use 3-4
R
delta removal check 3-32
release number
-r option, specification 3-29
change 3-2
change procedure 3-10
delta component 3-2
new file 3-29
protection 3-25, 3-25
rm command
file removal 3-6
rmdel command
delta removal 3-32
sccsdiff command
file comparison 3-33
sequence number
description 3-2
tab character
-n option, designation 3-31
user list
empty by default 3-24
login name addition 3-24
login name removal 3-25
protection feature 3-24
user name
list 3-24
v flag
new file use 3-17
what command
file information 3-33
write permission
delta removal 3-32
x-file
directory, location 3-3
naming procedure 3-3
permissions 3-3
temporary file copy 3-3
use 3-3
XENIX command

SCCS, source code control 1-2 *(continued)*
XENIX command *(continued)*
use precaution 3-27
z-file
lock file use 3-3
ownership 3-3
permissions 3-3
SID
components 3-2
delta printing use 3-22
setjmp function 7-10
setjmp.h file, described 7-1
signal function 7-1
signal.h file, described 7-1
Signals
catching 7-4
default action 7-3
delaying an action 7-8
described 7-1
disabling 7-2
redefining 7-4
restoring 7-3, 7-6
SIGHUP constant 7-1
SIGINT constant 7-1
SIGQUIT constant 7-1
SIG_DFL constant 7-1
SIG_IGN constant 7-1
to a child process 7-13
to background processes 7-12
with interactive programs 7-10
with multiple processes 7-12
with system functions 7-10
Software development
described 1-1
Source Code Control System See SCCS
strip
description 1-2
sum
description 1-2
Symbol
name list 1-2
removal 1-2
sync
description 1-3

T

Tags file
creation 1-2
tsort
description 1-2

X

XENIX file
identifying information 3-33
XENIX Operating system 1-1

Y

yacc
% token keyword
union member name association 6-32
%left keyword 6-21
union member name association 6-32
%left token
synonym 6-45
%nonassoc keyword 6-21
union member name association 6-32
%nonassoc token
synonyms 6-45
%prec keyword 6-22
synonym 6-45
%right keyword 6-21
union member name association 6-32
%right token
synonym 6-45
%token
synonym 6-45
%type keyword 6-32
0 key
endmarker token marker 6-10
-ly argument, library access 6-26
-v option
y.output file 6-13
0 character
grammar rules, avoidance 6-5
accept action See parser
accept simulation 6-30
action
0, negative number 6-30
conflict source 6-17
defined 6-6
error rules 6-24
form 6-45
global flag setting 6-29
input style 6-27, 6-27
invocation 6-1
location 6-7
nonterminating 6-8
parser See parser
return value 6-31
statement 6-6, 6-9
value in enclosing rules, access 6-30

yacc *(continued)*
  ampersand (&)
    bitwise AND operator 6-32
    desk calculator operator 6-32
  arithmetic expression
    desk calculator 6-32
    parsing 6-21
    precedence See precedence
  associativity
    arithmetic expression parsing 6-21
    grammar rule association 6-23
    recordation 6-23
    token attachment 6-21
  asterisk (*)
    desk calculator operator 6-32
  backslash (\)
    escape character 6-5
    percentage sign (%) substitution 6-45
  binary operator
    precedence 6-22
  blank character
    restrictions 6-4
  braces ({})
    action 6-9
    action statement enclosure 6-6
    action, dropping 6-45
    header file enclosure 6-31
  colon (:)
    identifier, effect 6-35
    punctuation 6-5
  comments
    location 6-4
  conflict
    associativity See associativity
    disambiguating rules 6-17, 6-18
    message 6-19
    precedence See precedence
    reduce/reduce conflict 6-17, 6-23, 6-23
    resolution, not counted 6-23
    shift/reduce conflict 6-17, 6-19, 6-23, 6-23
    source 6-17
  declaration section
    header file 6-31
    specification file component 6-4
Yacc
  description 1-1
  desk calculator specifications 6-32
    advanced features 6-38
    error recovery 6-38
    floating point interval 6-38
    scalar conversion 6-38
  dflag 6-29
  disambiguating rule 6-17
  disambiguating rules 6-18
  dollar sign ($)
    action significance 6-7
  empty rule 6-28

Yacc *(continued)*
  enclosing rules, access 6-30
  endmarker
    lookahead token 6-13
    parser input end 6-6
    representation 6-6
    token number 6-10
  environment 6-26
  error action See parser
  error token
    parser restart 6-24
    handling 6-23
    nonassociating implication 6-23
    parser restart 6-23
    simulation 6-30
    yyerrok statement 6-25
  escape characters 6-5
  external integer variable 6-27
  flag
    global flag See global flag
  floating point intervals See desk calculator
  global flag
    lexical analysis 6-29
  grammar rules 6-1, 6-2
    0 character avoidance 6-5
    advanced features 6-38
    ambiguity 6-16
    associativity association 6-23
    C code location 6-45
    empty rule 6-28
    error token 6-24
    format 6-4
    input style 6-27, 6-27
    left recursion 6-28
    left side repetition 6-5
    names 6-5
    numbers 6-20
    precedence association 6-23
    reduce action 6-12
    reduction 6-13
    rewrite 6-17
    right recursion 6-28
    specification file component 6-4
    value 6-7
  header file, union declaration 6-31
  historical features 6-44
  identifier
    input syntax 6-35
  if-else rule 6-18
  if-then-else construction 6-18
  input error detection 6-3
  input language 6-1
    style 6-27
    syntax 6-35
  keyword 6-21
    reservation 6-30
    union member name association 6-32
  left association 6-16

Yacc *(continued)*
left associative
reduce implication 6-23
left recursion 6-28
value type 6-32
lex
interface 5-2
lexical analyzer construction 6-11
lexical analyzer
context dependency 6-29
defined 6-1, 6-9
endmarker return 6-6
floating point constants 6-39
function 6-2
global flag examination 6-29
identifier analysis
lex 6-11
return value 6-31
scope 6-9
specification file component 6-4
terminal symbol See terminal symbol
token number agreement 6-9
lexical tie-in 6-29
library 6-26, 6-27
literal
defined 6-5
delimiting 6-44
length 6-44
lookahead token 6-11
clearing 6-25
error rules 6-24
LR(2) grammar 6-35
main program
minus sign (-)
desk calculator operator 6-32
names
composition 6-5
length 6-5
reference 6-4
token name See token name
newline character
restrictions 6-4
nonassociating
error implication 6-23
nonterminal name
input style 6-27
representation 6-5
nonterminal symbol 6-2
empty string match 6-5
location 6-6
name See nonterminal name
start symbol See start symbol
union member name association 6-32
octal integer
0 beginning 6-32
parser
accept action 6-13
accept simulation 6-30
actions 6-11

Yacc *(continued)*
parser *(continued)*
arithmetic expression 6-21
conflict See conflict
creation 6-21
defined 6-1
description 6-11
error action 6-13
error handling See error
goto action 6-12
initial state 6-15
input end 6-6
lookahead token 6-11
movement 6-11
names, yy prefix 6-9
nonterminal symbol See nonterminal
production failure 6-3
reduce action 6-11
restart 6-23
shift action 6-11
start symbol recognition 6-6
token number agreement 6-9
percentage sign (%)
action 6-9
desk calculator mod operator 6-32
header file enclosure 6-31
precedence keyword 6-21
specification file section separator 6-4
substitution 6-45
plus sign (+)
desk calculator operator 6-32
precedence
binary operator 6-22
change 6-22
grammar rule association 6-23
keyword 6-21
parsing function 6-21
recordation 6-23
token attachment 6-21
unary operator 6-22
program
specification file component 6-4
punctuation 6-5
quotation marks, double 6-44
quotation marks, single (")
literal enclosure 6-5
reduce action See parser
reduce command
number reference 6-20
reduce/reduce conflict 6-17, 6-23, 6-23

reduction conflict See reduce/reduce conflict

reduction conflict See shift/reduce conflict

reserved words 6-30
right association 6-16
right associative
shift implication 6-23

Yacc *(continued)*
  right recursion 6-28
  semicolon (;)
    input style 6-27
    punctuation 6-5
  shift action See parser
  shift command
    number reference 6-20
  shift/reduce conflict 6-17, 6-19, 6-23, 6-23

  simple-if rule 6-18
  slash (/)
    desk calculator operator 6-32
  specification file
    contents 6-4
    lexical analyzer inclusion 6-4
    sections separator 6-4
  specification files 6-3
  start symbol
    description 6-6
    location 6-6
  symbol synonyms 6-45
  tab character
    restrictions 6-4
  terminal symbol 6-2
  token name
    declaration 6-6
    input style 6-27
  token names 6-10
  token number 6-9
    agreement 6-9
    assignment 6-10
    endmarker 6-10
    associativity 6-21
    defined 6-1
    error token See error token
    names 6-4
    organization 6-1
    precedence 6-21
  unary operator
    precedence 6-22
  underscore sign (_)
    parser 6-15
  union
    copy 6-31
    declaration 6-31, 6-31
    header file 6-31
    name association 6-32
  unreachable statement 4-6
  value stack 6-31
    declaration 6-31
    floating point scalars, integers 6-38
    typing 6-31
    union See union
  vertical bar (|)
    bitwise OR operator 6-32
    desk calculator operator 6-32
    grammar rule repetition 6-5
    input style 6-27

Yacc *(continued)*
  y.output file 6-13
    parser checkup 6-23
  y.tab.c file 6-26
  y.tab.h file 6-31
  YYACCEPT 6-30
  yychar 6-26
  yyclearin statement 6-25
  yydebug 6-27
  yyerrok statement 6-25
  yyerror 6-26
  YYERROR 6-38
  yylex 6-26
  yyparse 6-26
    YYACCEPT effect 6-30
  YYSTYPE 6-32
  lint 4-11